

Introduction to PIC Programming

Baseline to Enhanced Mid-Range Architecture

by David Meiklejohn, Gooligum Electronics

Lesson 0: Recommended Training Environment

About PICs

“PIC” refers to an extensive family of microcontrollers manufactured by Microchip Technology Inc. – see www.microchip.com.

A microcontroller is a microprocessor which has I/O circuitry and peripherals built-in, allowing it to interface more or less directly with real-world devices such as lights, switches, sensors and motors. They simplify the design of logic and control systems, allowing complex (or simple!) behaviours to be designed into a piece of electronic or electromechanical equipment. They represent an approach which draws on both electronic design and programming skills; an intersection of what was once two disciplines, and is now called “embedded design”.

Modern microcontrollers make it very easy to get started. They are very forgiving and often need little external circuitry.

Among the most accessible are the PIC microcontrollers.

The range of PICs available is very broad – from tiny 6-pin 8-bit devices with just 16 bytes (!) of data memory which can perform only basic digital I/O, to 100-pin 32-bit devices with 512 kilobytes of memory and many integrated peripherals for communications, data acquisition and control.

One of the more confusing aspects of PIC programming for newcomers is that the low-end devices have entirely separate address and data buses for data and program instructions. When a PIC is described as being 8- or 16-bit, this refers to the amount of data that can be processed at once: the width of the data memory (registers in Microchip terminology) and ALU (arithmetic and logic unit).

The low-end PICs, which operate on data 8-bits at a time, are divided into four architectural families:

- **Baseline (12-bit instructions)**

These PICs are based on the original PIC architecture, going back to the 1970’s and General Instrument’s “Peripheral Interface Controller”. They are quite limited, but, within their limitations (such as having no interrupts), they are simple to work with.

Modern examples include the 6-pin 10F200, the 8-pin 12F509 and the 14-pin 16F506

- **Mid-Range (14-bit instructions)**

This is an extension of the baseline architecture, adding support for interrupts, more memory and peripherals, including PWM (pulse width modulation) for motor control, support for serial, I²C and SPI interfaces and LCD (liquid crystal display) controllers.

Modern examples include the 8-pin 12F629, the 20-pin 16F690 and the 40-pin 16F887

- **Enhanced Mid-Range (14-bit instructions)**

As the name suggests, this is an enhancement of the mid-range architecture – quite similar, but with additional instructions, simplified memory access (optimised for C compilers), and more memory, peripherals and speed.

Examples include the 8-pin 12F1501, the 14-pin 16F1824, and the 64-pin 16F1946.

- **High-end (16-bit instructions)**

Otherwise known as the 18F series, this architecture overcomes some of the limitations of the mid-range devices, providing for more memory (up to 128k program memory and almost 4k data memory) and advanced peripherals, including USB, Ethernet and CAN (controller area network) connectivity.

Examples include the 18-pin 18F1220, the 28-pin 18F2455 and the 80-pin 18F8520.

This can be a little confusing; the PIC18F series has 16-bit program instructions which operate on data eight bits at a time, and is considered to be an 8-bit chip.

The Gooligum Baseline, Mid-Range and Enhanced Mid-Range PIC Tutorials

The Gooligum tutorials introduce the baseline, mid-range and enhanced mid-range 8-bit PIC architectures, explaining the devices' internal structure, their ports (the pins used to interface with the real world) and common peripherals such as timers and analog-to-digital converters, using assembly language and C.

The tutorials are divided into a number of series (you don't have to do them all!):

- [Baseline Architecture and Assembly Language](#)
- [Programming Baseline PICs in C](#)
- [Mid-Range Architecture and Assembly Language](#)
- [Programming Mid-Range PICs in C](#)
- [Enhanced Mid-Range Architecture and Assembly Language](#)
- [Programming Enhanced Mid-Range PICs in C](#)

Some PIC tutorials focus on a single device, usually something fairly high-end, so that, by the end of the tutorials, you will have learned that device thoroughly and be well placed to learn other similar (or simpler) PICs easily, by studying the data sheets. And some tutorials note that most professional microcontroller development these days is done using C, so you might as well start by learning C.

While those approaches are perfectly valid, the philosophy behind the Gooligum tutorials has traditionally been that it's easiest to learn by starting with the least complex PICs first, without being distracted by a massive set of options (which will mostly be ignored at first), or having to say "do this, and we'll explain why later". And, for a thorough understanding of the PIC architecture, it is best to start by learning assembly language, because it's closer to the hardware – you can see exactly what is happening.

Thus, the [baseline assembly language series](#) begins with the simplest PIC devices, allowing the most basic concepts to be explained, one at a time, with more advanced devices being introduced as necessary.

Although the [baseline C series](#) recaps those explanations, it does not go into much detail – mostly just showing how the concepts from each assembly language lesson can be implemented in C.

Similarly, the [mid-range assembly language lessons](#) start with one of the simplest mid-range PICs and begin by recapping material from the baseline tutorials, with a focus on highlighting the differences and "what's new", before moving on to topics covering facilities not available in the simpler baseline PICs.

So, although it's possible to start with the [mid-range C tutorials](#), you will gain a better, deeper understanding by starting with baseline assembly and working your way up.

For those who want to follow that path, working your way through the baseline and then mid-range tutorials, follow-up series are available on migrating to the [enhanced mid-range architecture using assembly language](#) or [C](#). These series are intended for those who have completed the baseline and mid-range tutorials, highlighting the differences and introducing the new features of the enhanced mid-range architecture, without recapping explanations from the earlier lessons. These migration series act as a bridge from the mid-range tutorials to the new topics covers in the later enhanced mid-range lessons.

But, that's a long road to travel. The enhanced mid-range PICs are so capable and yet inexpensive that it makes little sense (unless you're chasing every cent to cost-reduce a volume product) to use baseline or mid-range PICs in new designs. Although starting with baseline PICs is an "easy" introduction to 8-bit PICs, it means learning a number of techniques which are not needed for enhanced mid-range devices.

For that reason, the [enhanced mid-range assembly language](#) series assumes no prior knowledge and does not reference the earlier baseline or mid-range lessons. It's a fresh start, designed for new students, who want to make a start with enhanced mid-range PICs.

And because C really is a very appropriate and popular choice for microcontroller development these days, the [enhanced mid-range C lessons](#) do not assume that you have completed the assembly language lessons, repeating most of the explanations¹.

Training / Development Environment

For PIC development, you'll need:

- A desktop or laptop PC, with a spare USB port.
Windows 7 (32- or 64-bit is ok) will give you the greatest choice of development environments and tools.
Linux and MacOS X are viable alternatives, with most of Microchip's development tools available for and supported on those platforms.
- Development software, including assembler and editor, preferably bundled in an integrated development environment (IDE) such as MPLAB X.
- A PIC programmer, to load your program into your PIC
- A prototyping environment, such as the [Gooligum Baseline and Mid-range PIC Training and Development Board](#), or simply a prototyping breadboard and your own supply of components, to allow you to build the example circuits in the tutorials.

And optionally:

- A C compiler.
The C tutorials use Microchip's XC8 compiler (running in "Free mode")² but you may still find the lessons helpful if you are using a different compiler.
- A hardware debugger (see below)

¹ of course, that does make it a little more repetitive for those who have completed the enhanced mid-range assembly language lessons – but it's not difficult to skim the duplicate material

² the baseline C lessons also use the free CCS compiler bundled with MPLAB 8.xx

PIC Programmers and Debuggers

There are many PIC programmers available, including some that you can build yourself.

Once upon a time, PICs could only be erased by shining UV light through a window on the chip (except for parts without a window, which could only be programmed once), and programmed by placing them into a special programmer.

These days, PICs use electrically erasable flash memory. They can be programmed without having to be taken out of the prototyping (or even production) environment, through a protocol called In-Circuit Serial Programming (ICSP). But instead of worrying about designing your circuit to accommodate the ICSP protocol, it can easier (especially for small PICs, where you may not have spare pins available to dedicate to the programming function) to remove the PIC from your circuit and place it into a development board or programming adapter connected to an ICSP programmer. A programming adapter is simply a minimal circuit which allows a PIC to be programmed by an ICSP programmer.

It's a really good idea to buy an ICSP programmer; you can use it with a development board or a programming adapter, while keeping the option of later using it with your own circuitry when you're ready for that.

An excellent PIC programmer to start with is Microchip's PICKit 3, shown on the right.

Although there are cheaper equivalents available³, the PICKit 3 is not expensive, available for around US\$45. And being a Microchip product, you can be sure that it will work with Microchip's development tools, and (importantly!) PICs.



The PICKit 3 can also work as a debugger, as long as the PIC you are using supports hardware debugging (meaning that it has special debug circuitry built in). Although these tutorials don't cover hardware debugging, it is a very useful facility to have available as you develop your own projects – it allows you to see exactly what the PIC is doing by stepping through your code an instruction at a time, or stopping at a particular location (a 'breakpoint'), and being able to see the contents of the PIC's internal registers and your program's variables. A hardware debugger is great for figuring out "what on Earth is that PIC doing?" And the great thing about buying a PICKit 3 as a programmer is that you get a capable hardware debugger as well, because they do that, too.

When you become more experienced, and/or work on bigger PICs, you may want to step up to a more capable (and of course more expensive) debugger, such as Microchip's ICD 3 or REAL ICE. But to start with, a PICKit 3 is ideal.

Development Software

Every PIC developer should have a copy of Microchip's MPLAB integrated development environment (IDE) – even if you primarily use a third-party *tool chain* (a set of development tools that work together).

It includes Microchip's assembler (MPASM), an editor, and a software simulator, which allows you to debug your application before committing it to the chip. Not long ago, a development environment as sophisticated as this would have cost thousands. But MPLAB is free, including support from Microchip, so there is no reason not to have it. Download it from www.microchip.com.

³ Such as the PICKit 2, described in the baseline and mid-range lessons. Although a capable programmer, it does not support enhanced mid-range PICs, and therefore the PICKit 2 is no longer recommended for use with these tutorials.

MPLAB directly supports the PICkit 3 as a programmer for pretty much every modern baseline, mid-range and enhanced mid-range PIC.

Microchip provides and supports two different “MPLAB” products.

For many years, MPLAB had been only available as a Windows application, and the latest versions are numbered 8.xx – the most recent at the time of writing (October 2013) is v8.92. We can refer to this as MPLAB 8. It’s very stable, easy to use, and has all the features we need.

However, Microchip has found it difficult to add additional features to MPLAB, or to meet requests to have it run on platforms other than Windows. Therefore, Microchip has developed a “next generation” replacement, called MPLAB X, which also runs on Linux and Mac OS X. MPLAB 8 is now effectively be retired, with all new development, including support for new tools, compilers and PICs, being for MPLAB X.

Therefore, although the older baseline and mid-range tutorials describe both MPLAB 8 and MPLAB X, the enhanced mid-range tutorials focus only on MPLAB X⁴.

MPLAB 8 includes a free copy of CCS’s PCB C compiler for Windows, which supports most baseline PICs, including those used in these tutorials. Although it’s now a little dated (at the time of writing, the version bundled with MPLAB was 4.073, while the latest commercially available version was 5.013), it remains useful and is used in the [baseline C tutorials](#).

The Microchip XC8 C compiler can be downloaded from www.microchip.com. It supports all of the baseline, mid-range and enhanced mid-range PICs, and is available for Windows, Linux and OS X. It can be used for free, when running in “Free mode”, but with most optimisations disabled – meaning that it generates much larger code than the paid-for commercial versions. However, code size doesn’t matter much for the small example programs in these tutorials, so the free version of XC8 is used in the [baseline](#), [mid-range](#) and [enhanced mid-range C tutorials](#).

Prototyping

If you use an ICSP programmer, then you’ll need a way to connect your PIC to it, for programming. You also need to be able to test your PIC in a real circuit, before building the final design.

One solution, satisfying both these purposes, is Microchip’s “PICkit 3 Low Pin Count (LPC) Demo Board”. It is available for around US\$26, including a PIC16F1829 and PIC18F14K22. Or you can buy a “PICkit 3 Starter Kit” bundle, including a PICkit 3 programmer, LPC Demo Board, PIC16F1829 and PIC18F14K22, for around US\$60. That’s excellent value; given that the MPLAB software is free, that’s everything you need to get started, including a programmer, demo board and PIC chips, for only US\$60!

It includes four LEDs, a pushbutton switch and a trimpot (variable resistor). Above this, a prototyping area is provided. The IC socket supports all of the modern (flash memory based) 8-, 14- and 20-pin baseline, mid-range and enhanced mid-range PICs. Note that it **does not** support the 6-pin 10F PICs, even when they are in 8-pin DIP package.

Most of the I/O lines are brought out to the 14-pin header on the side of the board, allowing it to be connected to another circuit. For example, a prototype circuit can be constructed on a breadboard and the power and signal lines connected back to the header on the LPC Demo Board. This arrangement allows you to develop a complex circuit with no need to remove the PIC from its socket for programming; the PICkit 3 can remain plugged into the LPC Demo Board during development.

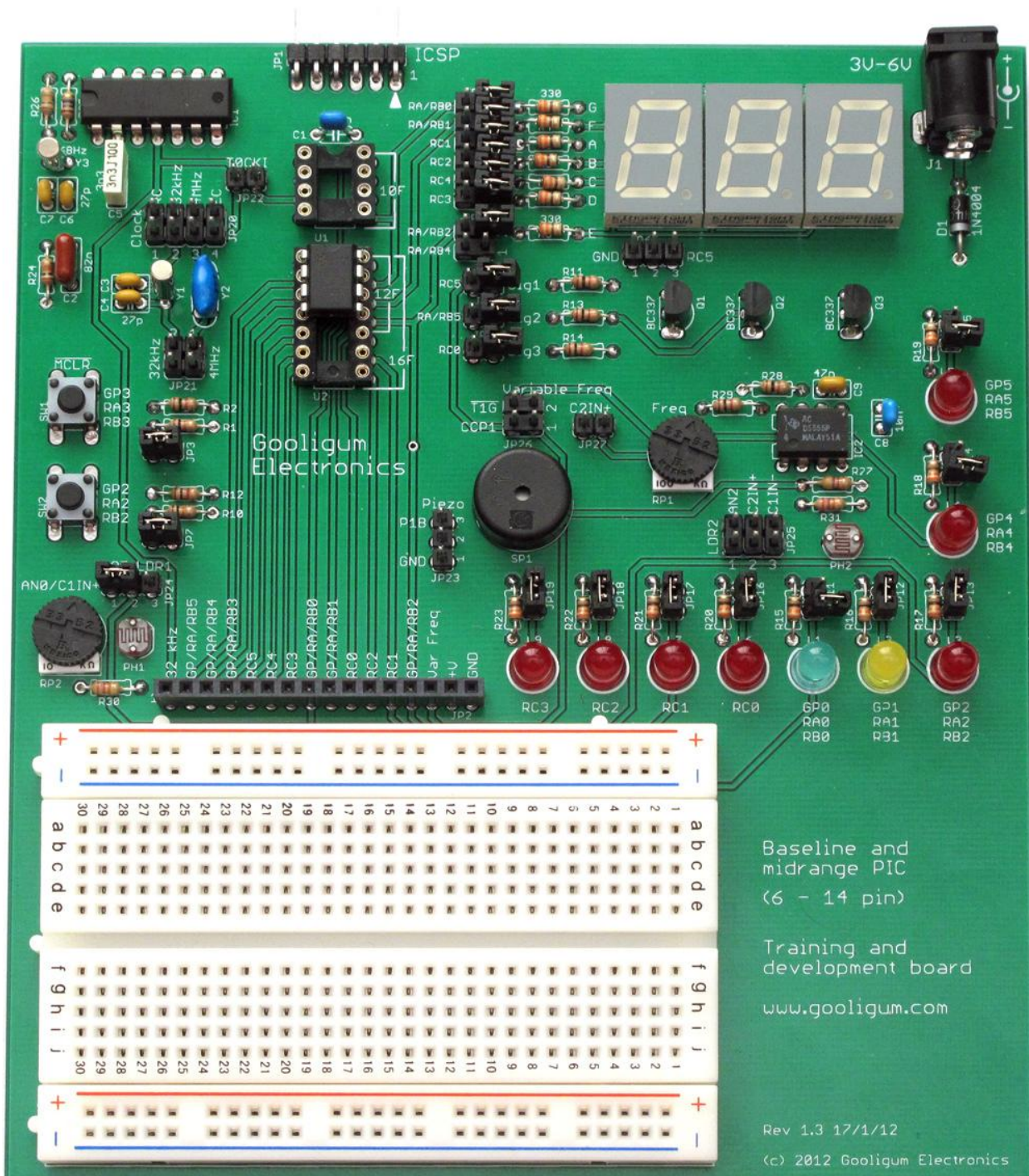
⁴ MPLAB 8 supports the devices used in the enhanced mid-range PIC tutorials and can be used with those lessons if you are already comfortable with MPLAB 8. However, it is unlikely to support future enhanced mid-range devices.

Unfortunately, the LEDs on the LPC Demo Board can only be used with 14- and 20-pin PICs, not the 8-pin devices. The board also doesn't come with jumpers installed; it's a good idea to add them, so that the LEDs and trimpot can be selectively disconnected, to avoid interference with the rest of your circuit.

Many of the tutorial lessons require the use of parts not included on the LPC Demo Board, such as photocells, crystal-driven oscillator circuits, and 7-segment LED displays. Although it is possible to build all of these circuits on a breadboard, connected to the LPC Demo Board, it is a little cumbersome to do so, for some of the more complex circuits.

And of course you need to obtain all the necessary parts.

To avoid these problems, we have developed a [training and development board](#), specifically for use with the baseline, mid-range and introductory enhanced mid-range tutorials, as shown below.



It works with a PICKit 3 programmer, and supports all 8- and 14-pin baseline, mid-range and enhanced mid-range PICs, as well as all 6-pin 10F devices (in an 8-pin DIP package). It is fully configurable using the provided jumpers, and comes with all of the hardware needed for the baseline and introductory mid-range and enhanced mid-range tutorials, including all the required PICs. Add-on parts kits, such as a motor-control kit, allow the board to be used with more topics covered in more advanced lessons.

Every PIC pin is brought out to the header at the bottom of the board, allowing the easy prototyping on the breadboard of circuits not provided by the onboard components, including circuits which require more than the 20 mA that a PICKit 3 can deliver, through the use of an external regulated DC power supply.

A number of other prototyping boards are available from various of sources, including Microchip. Some of these include more advanced peripherals, such as LCD displays, while others are intended to be an introduction to “mechatronics” (microcontroller-controlled robotics), and include motors, gears, etc. And some are intended to be general development boards, offering as much flexibility and expansion as possible. Most of these boards can of course be adapted for use with these tutorials.

However, the examples in these tutorials are intended to be used directly, without modification with the [Gooligum Baseline and Mid-range PIC Training and Development Board](#).

Recommendation

To make a start in PIC development, it’s difficult to do better than the low-cost combination of:

- PICKit 3 programmer
- Gooligum Baseline and Mid-Range PIC Training and Development Board
- MPLAB X integrated development environment

These tutorials assume that you are using that recommended combination. However, most of the lesson content is of course applicable to other development environments, including the Microchip Low Pin Count Demo Board, but you may need to modify the examples to work correctly in those environments.

Other than a PC, the only other thing you need is a PIC!

If you purchase the Gooligum training board, it will come with all the necessary PICs.

Otherwise, to follow all the lessons exactly, you will need one each of:

- PIC10F200 or PIC12F508
- PIC12F509
- PIC16F506
- PIC12F629
- PIC16F684
- PIC12F1501

It is possible to adapt the lessons to other baseline and mid-range PICs by reading the data sheets, but of course it’s easier to work with those listed here.

Good luck!!

Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 1: Light an LED

This initial exercise is the “Hello World!” of PIC programming.

The apparently straightforward task of simply making an LED connected to one of the output pins of a PIC light up – never mind flashing or anything else – relies on:

- Having a functioning circuit in a workable prototyping environment
- Being able to use a development environment; to go from text to assembled PIC code
- Being able to correctly use a PIC programmer to load the code into the PIC chip
- Correctly configuring the PIC
- Writing code that will set the correct pin to output a high or low (depending on the circuit)

If you can get an LED to light up, then you know that you have a development, programming and prototyping environment that works, and enough understanding of the PIC architecture and instructions to get started. It’s a firm base to build on.

Getting Started

For some background on PICs in general and details of the recommended development environment, see [lesson 0](#). Briefly, these tutorials assume that you are using a Microchip PICkit 2 or PICkit 3 programmer and either the [Gooligum Baseline and Mid-range PIC Training and Development Board](#) or Microchip’s Low Pin Count (LPC) Demo Board, with Microchip’s MPLAB 8 or MPLAB X integrated development environment. But it is of course possible to adapt these instructions to a different programmers and/or development boards.

We’re going to start with the simplest PIC of all – the PIC10F200, a 6-pin “baseline” device¹. It is only capable of simple digital I/O and does not include any advanced peripherals or interfaces. That makes it a good chip to start with; we’ll look at the additional features of more advanced PICs later.

In summary, for this lesson you should ideally have:

- A PC running Windows (XP, Vista or 7), with a spare USB port
- Microchip’s MPLAB 8 IDE software
- A Microchip PICkit 2 or PICkit 3 PIC programmer
- The Gooligum baseline training board
- A PIC10F200-I/P microcontroller (supplied with the Gooligum training board)

¹ If you are using Microchip’s LPC Demo Board, you will have to substitute a PIC12F508, because the LPC board does not support the 10F PIC family.

You could use Microchip's new MPLAB X IDE software, which runs under Linux and Mac OS X, as well as Windows, instead of MPLAB 8.

Note however that Microchip's Low Pin Count Demo Board does not support the 6-pin 10F PICs. If you are using the LPC demo board, you can substitute a 12F508 (the simplest 8-pin baseline PIC) instead of the 10F200, with only a minor change in the program code, which we'll highlight later.

The four LEDs on the LPC demo board don't work (directly) with 8-pin PICs, such as the 12F508. So, to complete this lesson, using an LPC demo board, you need to either add an additional LED and resistor to the prototyping area on your board, or use some solid core hook-up wire to patch one of the LEDs to the appropriate PIC pin, as described later.

This is one reason the Gooligum training board was developed to accompany these tutorials – if you have the Gooligum board, you can simply plug in your 10F or 12F PIC, and go.

Introducing the PIC10F200

When working with any microcontroller, you should always have on hand the latest version of the manufacturer's data sheet. You should download the current data sheet for the 10F200 from www.microchip.com.

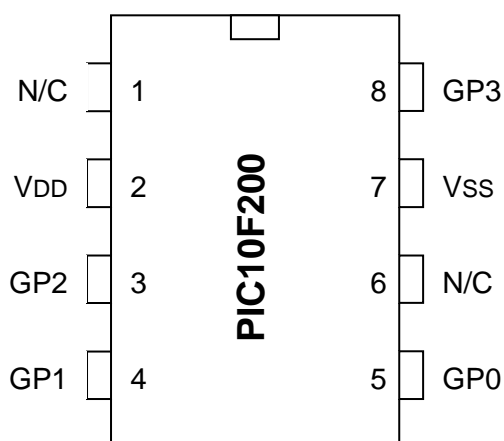
You'll find that the data sheet for the 10F200 also covers the 10F202, 10F204 and 10F206. These are essentially variants of the same chip. The differences are as follows:

Device	Program Memory (words)	Data Memory (bytes)	I/O pins	Comparators	Clock rate
10F200	256	16	4	0	4 MHz
10F202	512	24	4	0	4 MHz
10F204	256	16	4	1	4 MHz
10F206	512	24	4	1	4 MHz

The 10F202 and 10F206 have more memory, and the 10F204 and 10F206 include a comparator (used to compare analog signals – see [lesson 9](#)), but they are otherwise the same.

The 10F family are all 6-pin PICs, with only four pins available for I/O (input and output). They are typically used to implement simple functions, such as replacing digital logic, in space-constrained situations, so are generally used as tiny surface-mount devices.

However, they are also available in 8-pin DIP packages, as shown here.



These 8-pin DIP packages make prototyping easy, so we will use them in these lessons.

Note however, that although this is an 8-pin package, the PIC10F200 is still a 6-pin device. Only six of these pins are usable: the pins marked 'N/C' are not connected, and cannot be used.

VDD is the positive power supply.

VSS is the "negative" supply, or ground. All of the input and output levels are measured relative to VSS. In most circuits, there is only a single ground reference, considered to be at 0V (zero volts), and VSS will be connected to ground.

The power supply voltage (VDD, relative to VSS) can range from 2.0 V to 5.5 V.

This wide range means that the PIC's power supply can be very simple. Depending on the circuit, you may need no more than a pair of 1.5 V batteries (3 V total; less as they discharge).

Normally you'd place a capacitor, typically 100 nF, between VDD and VSS, close to the chip, to smooth transient changes to the power supply voltage caused by changing loads (e.g. motors, or something as simple as an LED turning on) or noise in the circuit. It is a good practice to place these "bypass capacitors" in any circuit beyond the simplest prototype stage, although you'll find that, particularly in a small battery-powered circuit, the PIC will often operate correctly without them. But figuring out why your PIC keeps randomly resetting itself is hard, while 100 nF capacitors are cheap, so include them in your designs!

The remaining pins, GP0 to GP3, are the I/O pins. They are used for digital input and output, except for GP3, which can only be an input. The other pins – GP0, GP1 and GP2 – can be individually set to be inputs or outputs.

PIC10F200 Registers

Address

00h	INDF
01h	TMR0
02h	PCL
03h	STATUS
04h	FSR
05h	OSCCAL
06h	GPIO
10h	General Purpose Registers
1Fh	
	OPTION
	W

8-bit PICs use a so-called Harvard architecture, where program and data memory is entirely separate.

In the 10F200, program memory extends from 000h to 0FFh (hexadecimal). Each of these 256 addresses can hold a separate 12-bit program instruction. User code starts by executing the instruction at 000h, and then proceeds sequentially from there – unless of course your program includes loops, branches or subroutines, which any real program will!

Microchip refers to the data memory as a "register file". If you're used to bigger microprocessors, you'll be familiar with the idea of a set of registers held on chip, used for intermediate values, counters or indexes, with data being accessed directly from off-chip memory. If so, you have some unlearning to do! The baseline PICs are quite different from mainstream microprocessors. The only memory available is the on-chip "register file", consisting of a number of registers, each 8 bits wide. Some of these are used as general-purpose registers for data storage, while others, referred to as special-function registers, are used to control or access chip features, such as the I/O pins.

The register map for the 10F200 is shown at left. The first seven registers are special-function, each with a specific address. They are followed by sixteen general purpose registers, which you can use to store program variables such as counters. Note that there are no registers from 07h to 0Fh on the 10F200.

The next two registers, TRIS and OPTION, are special-function registers which cannot be addressed in the usual way; they are accessed through special instructions.

The final register, W, is the working register. It's the equivalent of the 'accumulator' in some other microprocessors. It's central to the PIC's operation. For example, to copy data from one general purpose register to another, you have to copy it into W first, then copy from W to the destination. Or, to add two numbers, one of them has to be in W. W is used a lot!

It's traditional at this point to discuss what each register does. But that would be repeating the data sheet, which describes every bit of every register in detail. The intention of these tutorials is to only explain what's needed to perform a given function, and build on that. We'll start with the I/O pins.

PIC10F200 Input and Output

As mentioned above, the 10F200 has four I/O pins: GP0, GP1 and GP2, which can be used for digital input and output, plus GP3, which is input-only.

Taken together, the four I/O pins comprise the general-purpose I/O *port*, or GPIO port.

If a pin is configured as an output, the output level is set by the corresponding bit in the GPIO register. Setting a bit to '1' outputs a high voltage² on the corresponding pin; setting it to '0' outputs a low voltage³.

If a pin is configured as an input, the input level is represented by the corresponding bit in the GPIO register. If the voltage on an input pin is high⁴, the corresponding bit reads as '1'; if the input voltage is low⁵, the corresponding bit reads as '0':

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
GPIO					GP3	GP2	GP1	GP0

The TRIS register controls whether a pin is set as an input or output:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TRIS						GP2	GP1	GP0

To configure a pin as an input, set the corresponding bit in the TRIS register to '1'. To make it an output, clear the corresponding TRIS bit to '0'.

Why is it called 'TRIS'? Each pin (except GP3) can be configured as one of three states: high-impedance input, output high, or output low. In the input state, the PIC's output drivers are effectively disconnected from the pin. Another name for an output that can be disconnected is '*tri-state*' – hence, TRIS.

Note that bit 3 of TRIS is greyed-out. Clearing this bit will have no effect, as GP3 is always an input.

The default state for each pin is 'input'; TRIS is set to all '1's when the PIC is powered on or reset.

When configured as an output, each I/O pin on the 10F200 can source or sink (i.e. current into or out of the pin) up to 25 mA – enough to directly drive an LED.

In total, the I/O port can source or sink up to 75 mA.

So, if you were driving four LEDs, and it is possible for all to be on at once, you should limit the current in each LED to 18 mA, so that the total for the port will never be more than 75 mA, even though each pin can supply up to 25 mA on its own.

PICs are tough devices, and you may get away with exceeding these limits – but if you ignore the absolute maximum ratings specified in the data sheet, you're on your own. Maybe your circuit will work, maybe not. Or maybe it will work for a short time, before failing. It's better to follow the data sheet...

² a 'high' output will be close to the supply voltage (VDD), for small pin currents

³ a 'low' output is less than 0.6 V, for small pin currents

⁴ the threshold level depends on the power supply, but a 'high' input is any voltage above 2.0 V, given a 5 V supply

⁵ a 'low' input is anything below 0.8 V, given a 5 V supply – see the data sheet for details of each of these levels

Introducing the PIC12F508

If you're using the Microchip Low Pin Count Demo Board, you can't use a PIC10F200, because that board doesn't support the 10F family.

The simplest baseline PIC that you can use with the LPC demo board is the 8-pin 12F508.

As with the 10F200, the data sheet for the 12F508 also covers some related variants: in this case the 12F509 and 16F505. The differences are as follows:

Device	Program Memory (words)	Data Memory (bytes)	Package	I/O pins	Clock rate (maximum)
12F508	512	25	8-pin	6	4 MHz
12F509	1024	41	8-pin	6	4 MHz
16F505	1024	72	14-pin	12	20 MHz

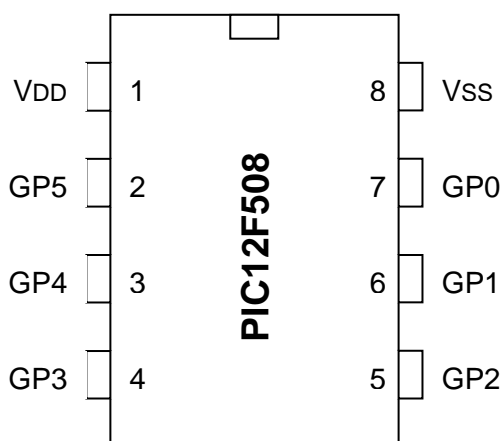
The 12F509 has more memory than the 12F508, but is otherwise identical. The 16F505 adds extra I/O pins, some more data memory, and can run at a higher speed (if driven by an external clock or crystal).

The 12F508 is essentially a 10F200 with more pins and memory.

It has six I/O pins (instead of four), labelled GP0 to GP5, in an 8-pin package. Like the 10F200, each pin can be configured as a digital input or output, except GP3, which is input-only.

The 12F508 has twice as much program memory as the 10F200 (512 words instead of 256), extending from 000h to 1Fh.

The register map, shown on the right, is nearly identical to that of the 10F200. The only difference is that the 12F508 has 25 general purpose registers, instead of 16, with no 'gaps' in the memory map.



PIC12F508 Registers

Address

00h	INDF	
01h	TMR0	
02h	PCL	
03h	STATUS	
04h	FSR	
05h	OSSCAL	
06h	GPIO	
07h	General Purpose Registers	
1Fh		
		TRIS
		OPTION
		W

Like the 10F200, the 12F508's six pins are mapped into the GPIO register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
GPIO			GP5	GP4	GP3	GP2	GP1	GP0

And the pin directions (input or output) are controlled by corresponding bits in the TRIS register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TRIS			GP5	GP4		GP2	GP1	GP0

Note that bit 3 is greyed out. That's because, as with the 10F200, GP3 is always an input.

Most other differences between the 12F508 and 10F200 concern the processor oscillator (or clock, governing how fast the device runs) configurations, which we'll look at in [lesson 7](#).

Example Circuit

We now have enough background information to design a circuit to light an LED.

We'll need a regulated power supply, let's assume 5 V, connected to VDD and VSS. And remember that we should add a bypass capacitor, preferably a 100 nF (or larger) ceramic, across it.

We'll also need an LED of course, and a resistor to limit the current.

Although the PIC10F200 or PIC12F508 can supply up to 25 mA from a single pin, 10 mA is more than enough to adequately light most LEDs. With a 5 V supply and assuming a red or green LED with a forward voltage of around 2 V, the voltage drop across the resistor will be around 3 V.

Applying Ohm's law, $R = V / I = 3 \text{ V} \div 10 \text{ mA} = 300 \Omega$. Since precision isn't needed here (we only need "about" 10 mA), it's ok to choose the next highest "standard" E12 resistor value, which is 330 Ω . It means that the LED will draw less than 10 mA, but that's a good thing, because, if we're going to use a PICKit 2 or PICKit 3 to power the circuit, we need to limit overall current consumption to 25 mA, because that is the maximum current those programmers can supply.

Finally, we need to connect the LED to one of the PIC's pins.

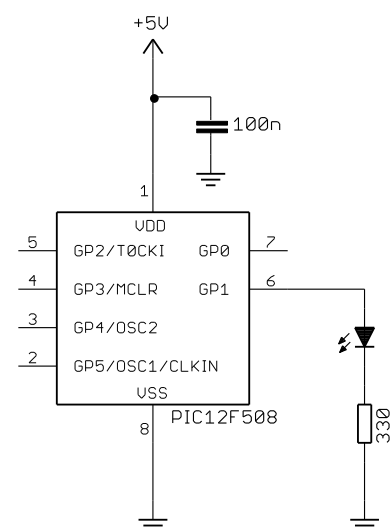
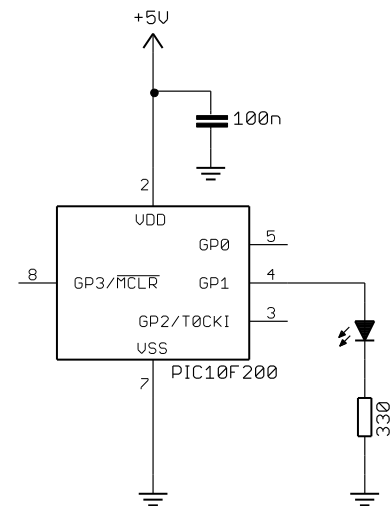
We can't choose GP3, because it's input-only.

If you're using the Gooligum training board, you could choose any of the other pins, but if you use the Microchip LPC Demo Board to implement the circuit, it's not a good idea to use GP0, because it's connected to a trimpot on the LPC demo board, which would divert current from the LED.

So, we'll use GP1, giving the circuits (10F200 and 12F508 versions) shown on the right.

Simple, aren't they? Modern microcontrollers really do have minimal requirements.

Of course, some connections are also needed for the ICSP (programmer) signals. These will be provided by your development board, unless you are building the circuit yourself. But the circuit as shown here is all that is needed for the PIC to run, and light the LED.



Gooligum training and development board instructions

If you have the Gooligum training board, you should use it to implement the first (10F200) circuit.

Plug the PIC10F200 into the 8-pin IC socket marked '10F'.⁶

Connect a shunt across the jumper (JP12) on the LED labelled 'GP1', and ensure that every other jumper is disconnected.

Plug your PICKit 2 or PICKit 3 programmer into the ICSP connector on the training board, with the arrow on the board aligned with the arrow on the PICKit, and plug the PICKit into a USB port on your PC.

The PICKit 2 or PICKit 3 can supply enough power for this circuit, so there is no need to connect an external power supply.

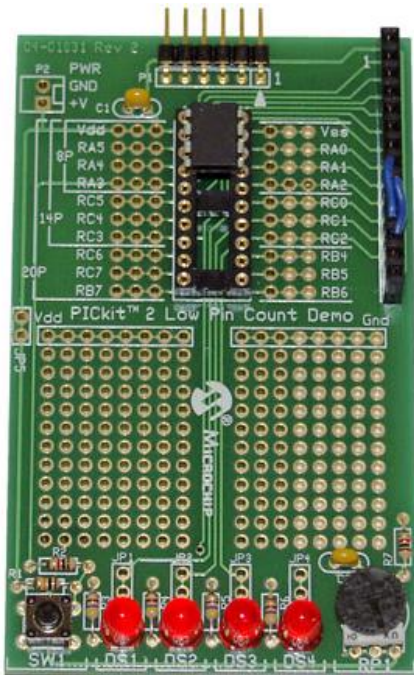
Microchip Low Pin Count Demo Board instructions

If you are using Microchip's LPC Demo Board, you'll need to take some additional steps.

Although the board provides four LEDs, they cannot be used directly with a 12F508 (or any 8-pin PIC), because they are connected to DIP socket pins which are only used with 14-pin and 20-pin devices.

However, the circuit can be readily built by adding an LED, a 330 Ω resistor and a piece of wire to the LPC Demo Board, as illustrated on the right.

In the pictured board, a green LED is wired to GP1 (labelled 'RA1') and a red LED to GP2 (labelled 'RA2'); we'll use both LEDs in later lessons. Jumper blocks have been added so that these LEDs can be easily disconnected from the PIC, to facilitate prototyping other circuits. These jumpers are wired in series with each LED.



Note that on the LPC Demo Board, the pins are labelled 'RA1', 'RA2', etc., because that is the nomenclature used on the larger 20-pin PICs, such as the 16F690. They correspond to the 'GP' pins on the 12F508 – simply another name for the same thing.

If you prefer not to solder components onto your demo board, you can use the LEDs on the board, labelled 'DS1' to 'DS4', by making connections on the 14-pin header on the right of the demo board, as shown on the left. This header makes available all the 12F508's pins, GP0 – GP5 (labelled 'RA0' to 'RA5'), as well as power (+5 V) and ground. It also brings out the additional pins, labelled 'RC0' to 'RC5', available on the 14-pin devices.

The LEDs are connected to the pins labelled 'RC0' to 'RC3' on the IC socket, via 470 Ω resistors (and jumpers, if you choose to install them). 'DS1' connects to pin 'RC0', 'DS2' to 'RC1', and so on.

⁶ Ensure that no device is installed in the 12F/16F socket – you can only use one PIC at a time in the training board.

So, to connect LED ‘DS2’ to pin GP1, simply connect the pin labelled ‘RA1’ to the pin labelled ‘RC1’, which can be done by plugging a short piece of solid-core hook-up wire into pins 8 and 11 on the 14-pin header.

Similarly, to connect LED ‘DS3’ to pin GP2, simply connect header pins 9 and 12.

That’s certainly much easier than soldering, so why bother adding LEDs to the demo board? The only real advantage is that, when using 14-pin and 20-pin PICs later, you may find it useful to have LEDs available on RA1 and RA2, while leaving RC0 – RC3 available to use, independently. In any case, it is useful to leave the 14-pin header free for use as an expansion connector, to allow you to build more complex circuits, such as those found in the later tutorial lessons: see, for example, [lesson 8](#).

Time to move on to programming!

Development Environment

You’ll need Microchip’s MPLAB Integrated Development Environment (MPLAB IDE), which you can download from www.microchip.com.

As discussed in [lesson 0](#), MPLAB comes in two varieties: the older, established, Windows-only MPLAB 8, and the new multi-platform MPLAB X. If you are running Windows, it is better to use MPLAB 8 as long as Microchip continues to support it, because it is more stable and in some ways easier to use. It also allows us to use all the free compilers referenced in the [C tutorial series](#). However, it is clear that Microchip’s future development focus will be on MPLAB X, and that it will become the only viable option. So in these tutorials we will look at how to install and use both MPLAB IDEs.

MPLAB 8.xx

When installing, if you choose the ‘Custom’ setup type, you should select as a minimum:

- ✓ Microchip Device Support
 - ✓ 8 bit MCUs (all 8-bit PICs, including 10F, 12F, 16F and 18F series)
- ✓ Third Party Applications
 - ✓ CCS PCB Full Install (C compiler for baseline PICs)
- ✓ Microchip Applications
 - ✓ MPASM Suite (the assembler)
 - ✓ MPLAB IDE (the development environment)
 - ✓ MPLAB SIM (software simulator – extremely useful!)
 - ✓ PICKit 2
 - or PICKit 3 Programmer/Debugger

It’s worth selecting the CCS PCB C compiler, since it will then be properly integrated with MPLAB; we’ll see how to use it later, in the C tutorial series.

You may need to restart your PC.

You can then run MPLAB IDE.

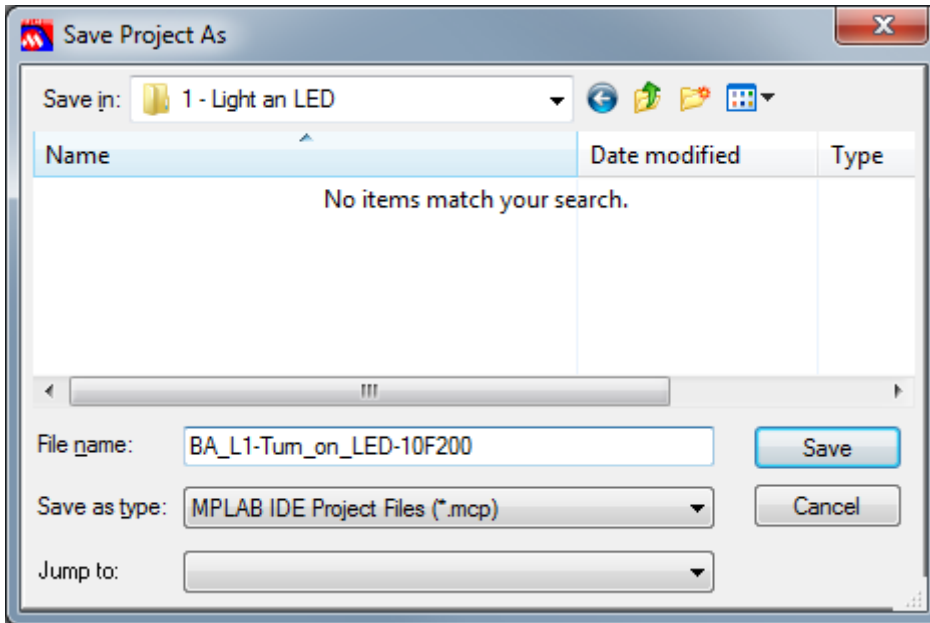
Creating a New Project

To start a new project, you should run the project wizard (Project → Project Wizard...).

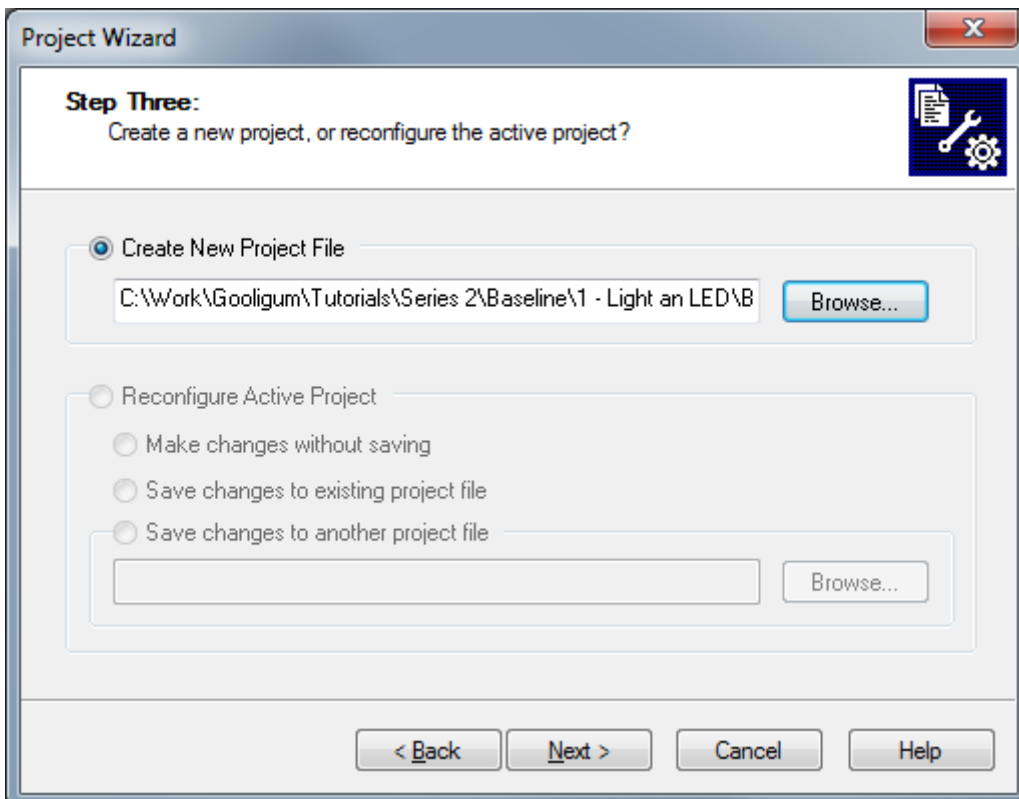
First, you'll be asked to select a device: in our case, the PIC10F200 or PIC12F508.

Then select MPASM as the active toolsuite, which tells MPLAB that this is an assembler project. Don't worry about the toolsuite contents and location; if you're working from a clean install, they'll be correct.

Next, select "Create New Project File" and browse to where you want to keep your project files, creating a new folder if appropriate. Then enter a descriptive file name and click on "Save", as shown:



You should end up back in the Project Wizard window, shown below.



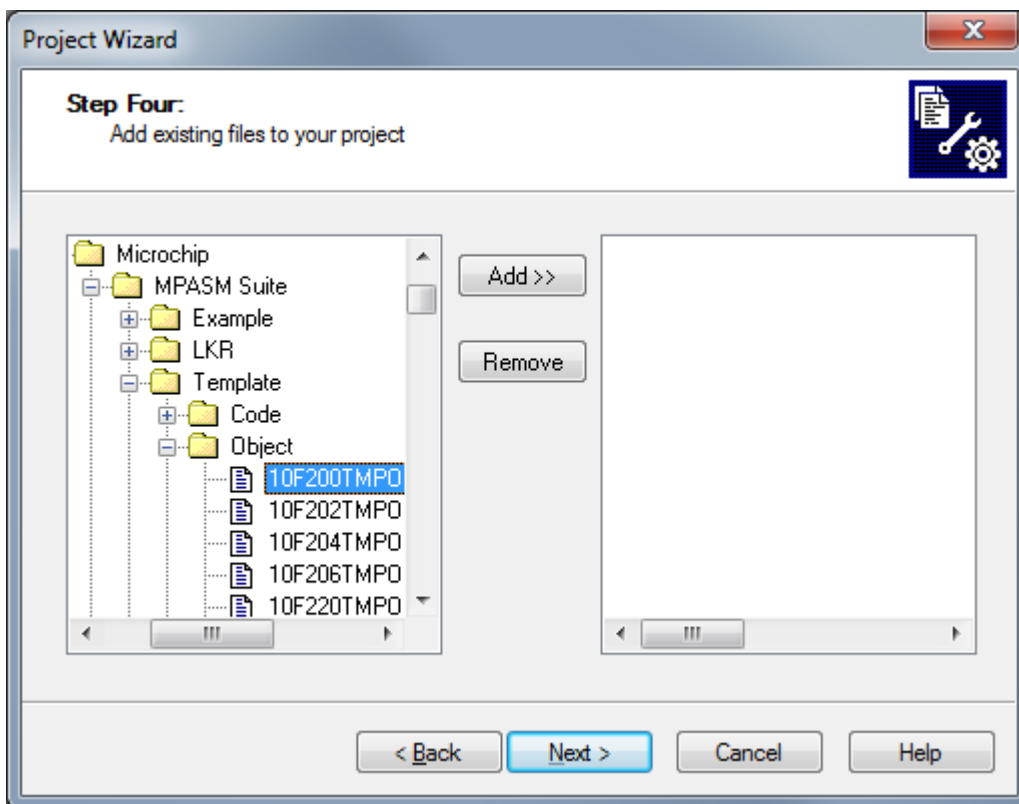
Microchip supplies templates you can use as the basis for new code. It's a good idea to use these until you develop your own. Step 4 of the project wizard allows you to copy the appropriate template into your project.

When programming in PIC assembler, you have to choose whether to create absolute or relocatable code. Originally, only absolute mode was supported, so most of the older PIC programming resources will only refer to it. In absolute mode, you specify fixed addresses for your code in program memory, and fixed addresses for your variables in data memory. That's ok for small programs, and seems simple to start with (which is another reason why many guides for beginners only mention absolute mode). But as you start to build larger applications, perhaps making use of reusable modules of previously-written code, and you start to move code from one PIC chip to another, you'll find that absolute mode is very limiting.

Relocatable code relies on a *linker* to assign object code (the assembled program instructions) and variables to appropriate addresses, under the control of a script specific to the PIC you're building the program for. Microchip supplies linker scripts for every PIC; unless you're doing something advanced, you don't need to touch them – so we won't look at them. Writing relocatable code isn't difficult, and it will grow with you, so that's what we'll use.

The templates are located under the 'Template' directory, within the MPASM installation folder (usually 'C:\Program Files\Microchip\MPASM Suite'⁷). The templates for absolute code are found in the 'Code' directory, while those for relocatable code are found in the 'Object' directory. Since we'll be doing relocatable code development, we'll find the templates we need in '<MPASM Suite>\Template\Object'.

In the left hand pane, navigate to the '<MPASM Suite>\Template\Object' directory and select the template for whichever PIC you are using: '10F200TMPO.ASM' or '12F208TMPO.ASM'.

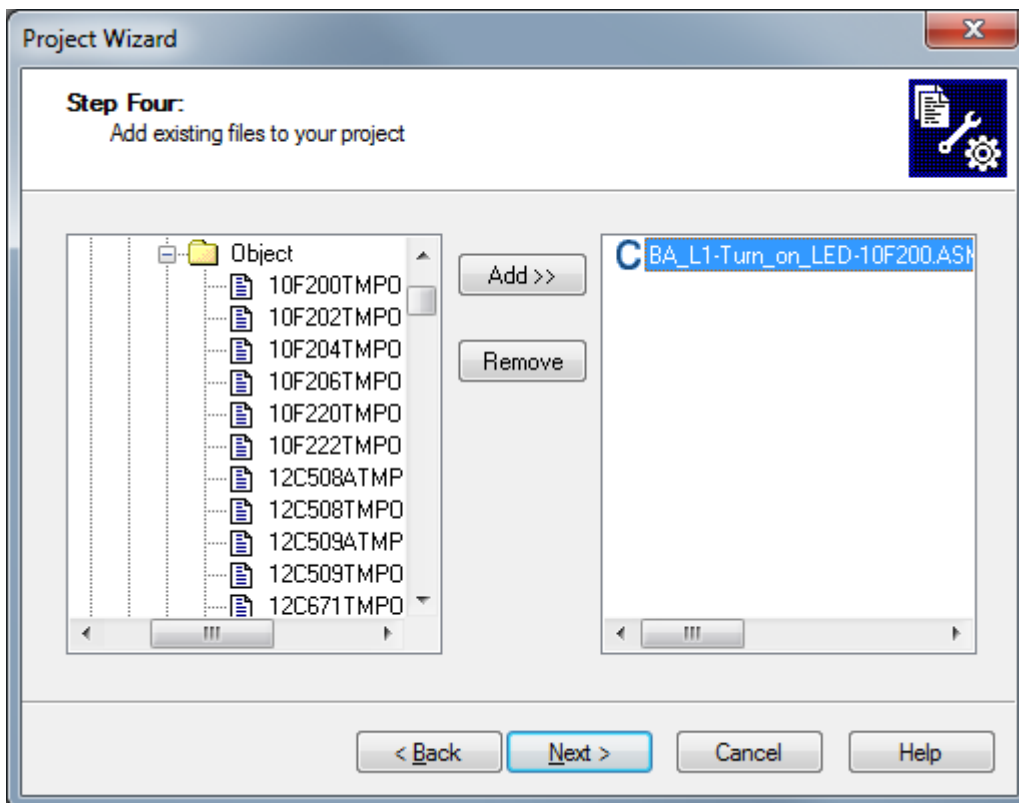


⁷ On a 64-bit version of Windows, the MPASM folder will be under 'Program Files (x86)'

When you click on the “Add>>” button, the file should appear in the right hand pane, with an “A” to the left of it. The “A” stands for “Auto”. In auto mode, MPLAB will guess whether the file you have added should be referenced via a *relative* or *absolute* path. Relative files are those that should move with the project (if it is ever moved or copied). Absolute files should always remain in a fixed location; they don’t belong specifically to your project and are more likely to be shared with others. Clicking the “A” will change it to “U”, indicating a “User” file which should be referenced through a relative path, or “S”, indicating a “System” file which should have an absolute reference.

We don’t want either; we need to copy the template file into the project directory, so click on the “A” until it changes to “C”, for “Copy”. When you do so, it will show the destination file name next to the “C”. Of course, you’re not going to want to call your copy ‘10F200TMPO.ASM’, so click the file name and rename it to something more meaningful, like ‘BA_L1-Turn_on_LED-10F200.asm’.

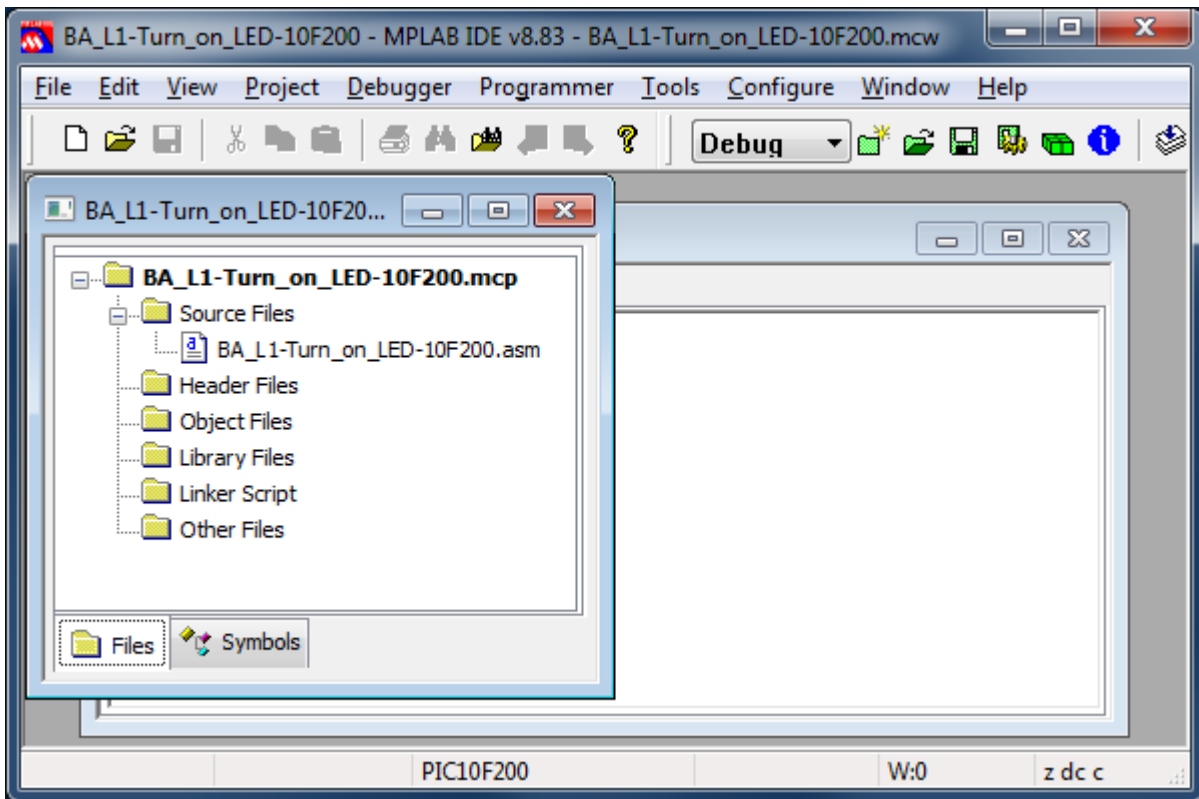
The window should now look similar to that shown below, with your (renamed) copy of the template file selected in the right hand pane:



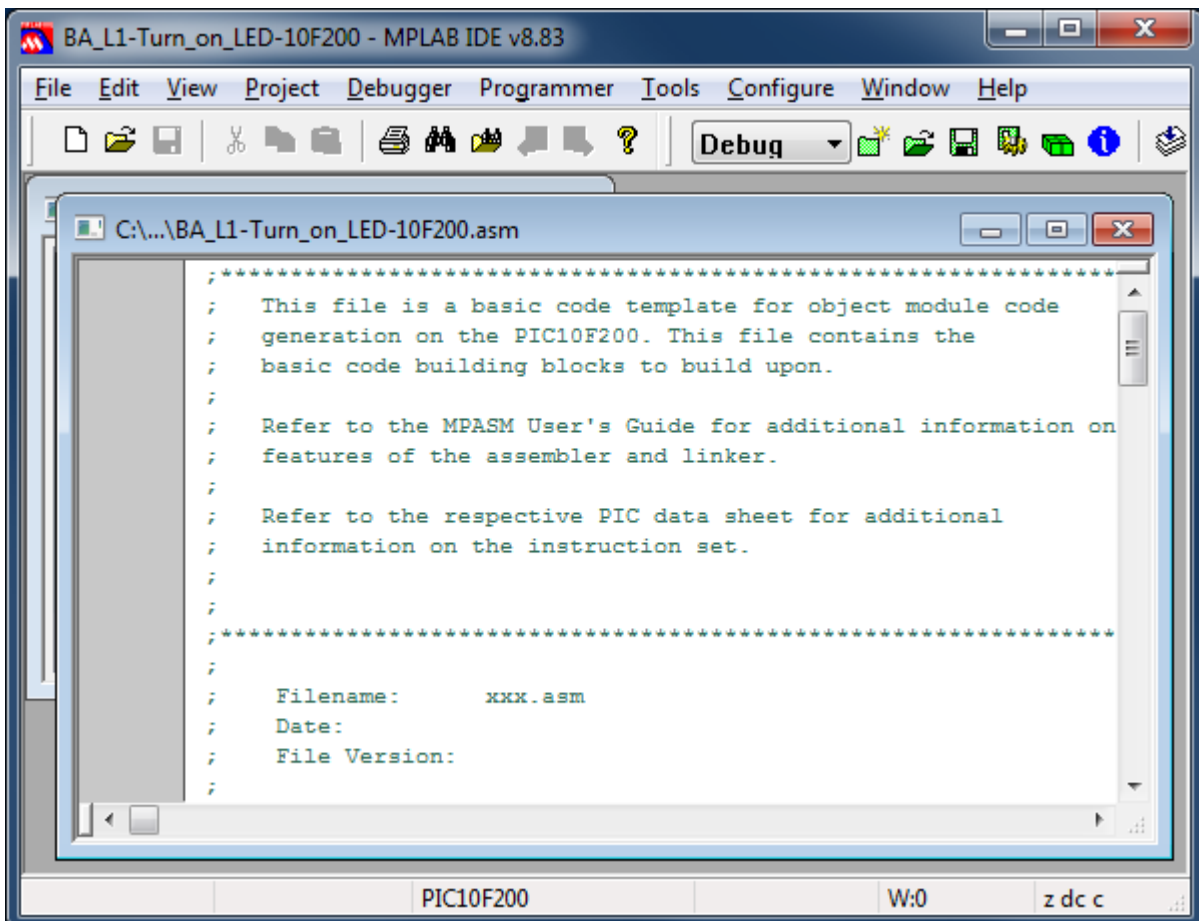
After you click “Next” and then “Finish” on the final Project Wizard window, you will be presented with an empty MPLAB workspace. You may need to select View → Project to see the project window, which shows your project and the files it comprises as a tree structure.

For a small, simple project like this, the project window will show only the assembler source file (.asm).

Your MPLAB IDE window should be similar to that illustrated at the top of the next page.




To get started writing your program, double-click your .asm source file. You'll see a text editor window open; finally you can see some code!



The MPLAB text editor is aware of PIC assembler (MPASM) syntax and will colour-code text, depending on whether it's a comment, assembler directive, PIC instruction, program label, etc. If you right-click in the editor window, you can set editor properties, such as auto-indent and line numbering, but you'll find that the defaults are quite usable to start with.

We'll take a look at what's in the template, and then add the instructions needed to turn on the LED.

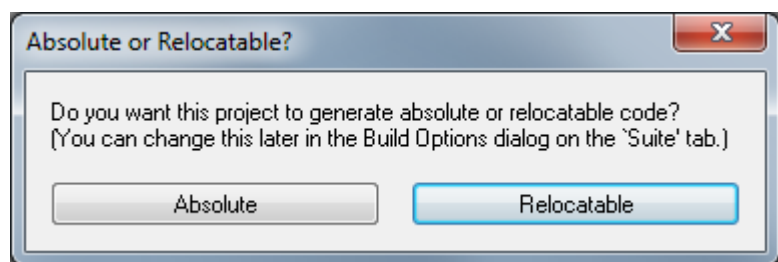


But first it's a good idea to save your new project (Project → Save Project, or click on , or simply exit MPLAB and click 'Yes', when asked if you wish to save the workspace).

When you re-open your project in MPLAB, which you can easily do by double-clicking the '.mcp' project file in your project directory, you may be asked if you are developing absolute or relocatable code.

You should choose 'Relocatable'.

Later, after building your project (see below), you won't be asked this question. But if ever see this prompt, just choose 'Relocatable'.



MPLAB X

You should download the MPLAB X IDE installer for your platform (Windows, Linux or Mac) from the MPLAB X download page at www.microchip.com, and then run it.

Unlike MPLAB 8, there are no installation options (other than being able to choose the installation directory). It's an "all or nothing" installer, including support for all of Microchip's PIC MCUs and development tools.

You can then run MPLAB X IDE.

Creating a New Project

When you first run MPLAB X, you will see the "Learn & Discover" tab, on the Start Page.

To start a new project, you should run the New Project wizard, by clicking on 'Create New Project'.

In the first step, you need to specify the project category. Choose 'Standalone Project'.

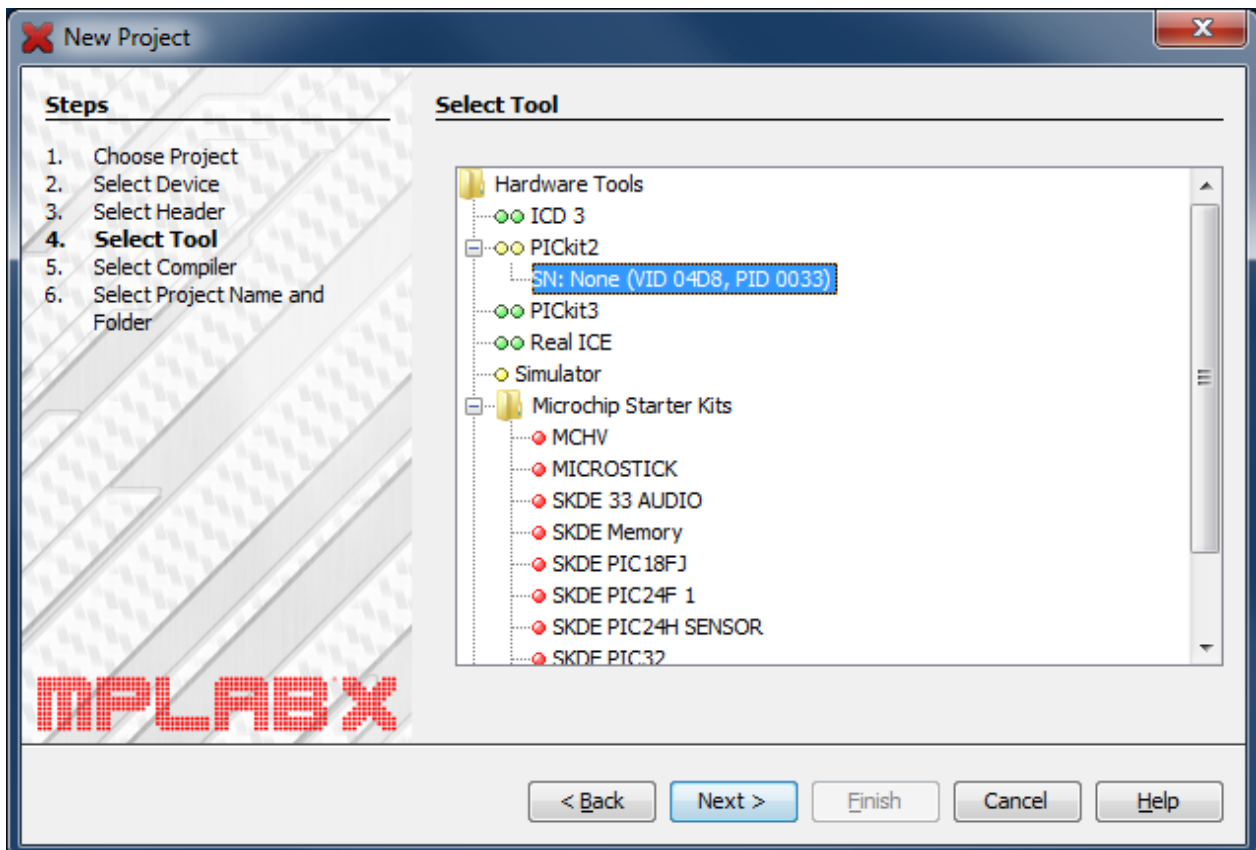
Next, select the PIC family and device.

In our case, we need 'Baseline 8-bit MCUs' as the family, and either the PIC10F200 or PIC12F508 as the device.

The third step allows you to optionally select a debug header. This is a device used to facilitate hardware debugging (see explanation in [lesson 0](#)), especially for PICs (such as the baseline devices we are using) which do not include internal hardware to support debugging. If you are just starting out, you are unlikely to have one of these debug headers, and you don't need one for these tutorials. So, you should not select a header. Just click 'Next'.

The next step is to select the tool you will use to program your PIC.

First, you should plug in the programmer (e.g. PICkit 2 or PICkit 3) you intend to use. If it is properly connected to your PC, with a functioning device driver⁸, it will appear in the list of hardware tools, and you should select it, as shown:

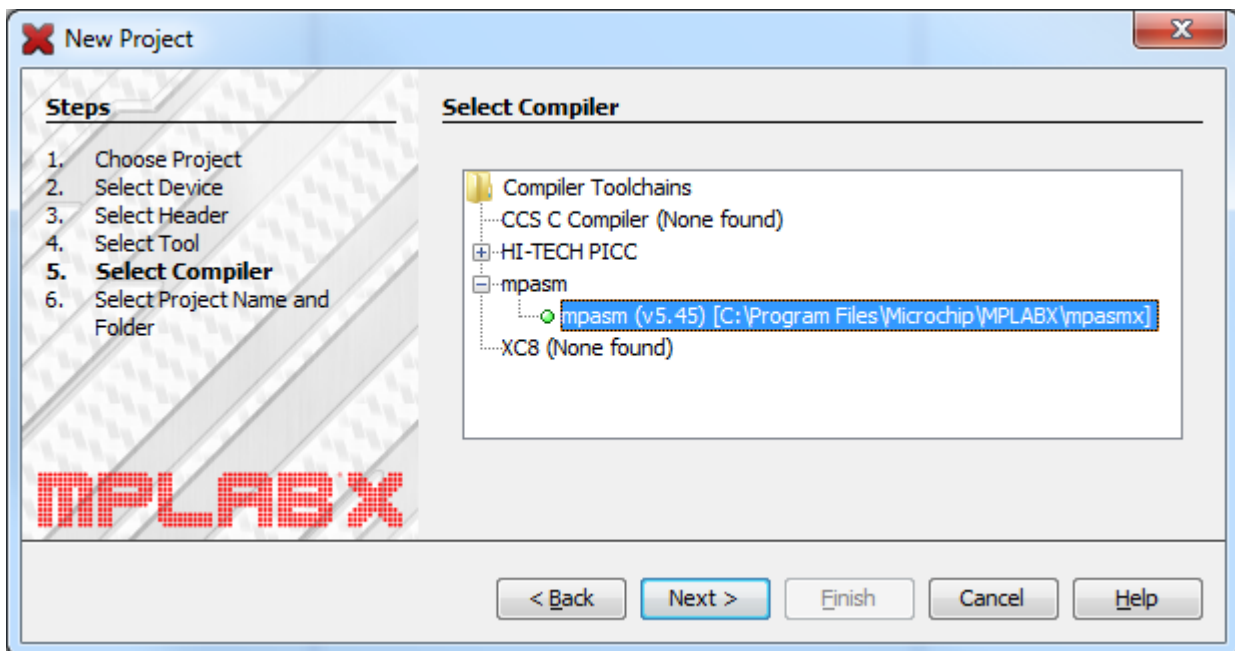


In this case, a PICkit 2 is connected to the PC.

If you have more than one programmer plugged in (including more than one of the same type, such as two PICkit 3s), they will all appear in this list, and you should select the specific one you intend to use for this project – you may need to check the serial number. Of course, you probably only have one programmer, so your selection will be easy.

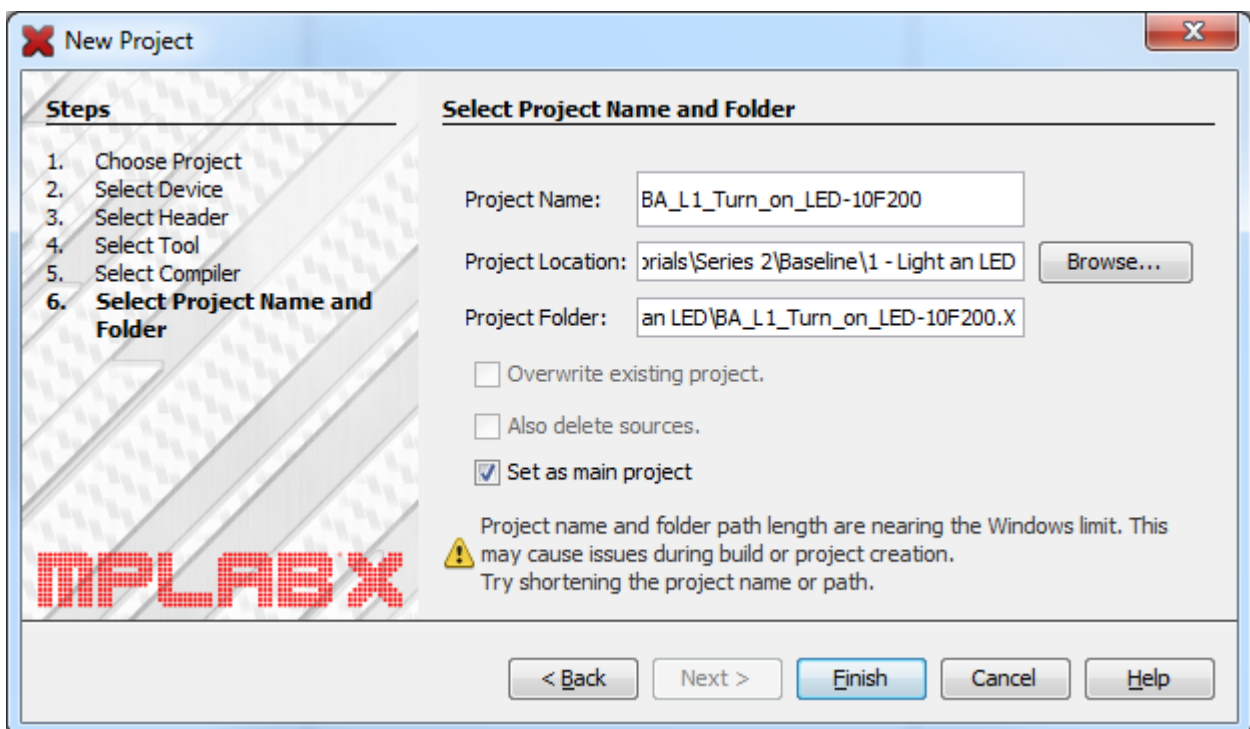
⁸ There is no need to install a special device driver for the PICkit 2 or PICkit 3; they work “out of the box”.

After selecting the hardware tool, you select the compiler (or, in our case, assembler) you will be using:



To specify that we will be programming in assembler, select the 'mpasm' option.

Finally, you need to specify your project's location, and give it a name:



MPLAB X creates a folder for its files, under the main project folder.

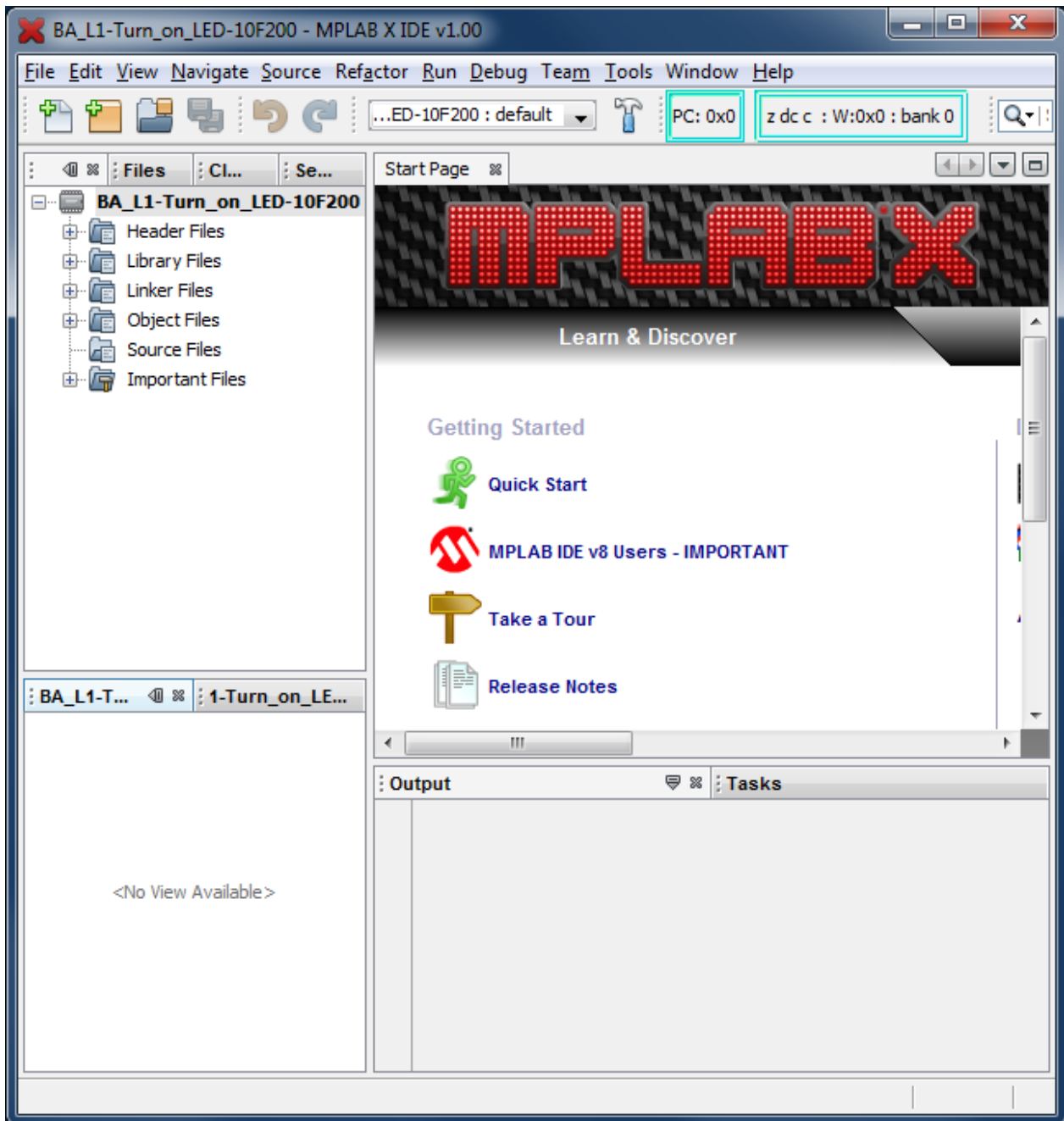
For example, in the environment used to develop these tutorials, all the files related to this lesson, including schematics and documentation, are placed in a folder named '1 - Light an LED', which is the "Project Location" given above. MPLAB X then creates a separate folder for the PIC source code and other files related to this project, in a subfolder that, by default, has the same name as the project, with a

‘.X’ on the end. If you wish, you can remove the ‘.X’ extension from the project folder, before you click on ‘Finish’.

Note the warning about project name and folder path length. To avoid possible problems, it’s best to use shorter names and paths, when using Windows, although in this case it’s actually ok.

Since this is the only project we’re working on, it doesn’t make much difference whether you select ‘Set as main project’; this is something that is more useful when you are working with multiple projects.

After you click “Finish”, your project will appear in the project window, and your workspace should look something like this:



It is usually best to base your new program on an existing template (which could be a similar program that you developed earlier).

As explained in the MPLAB 8 instructions above, a set of templates for either absolute or relocatable code development is provided with MPASM. These are located under the ‘templates’ directory, within the MPASM installation directory, which, if you are using a 32-bit version of Windows, will normally be ‘C:\Program Files\Microchip\MPLABX\mpasmx’⁹. As before, the templates for absolute code are found in the ‘Code’ directory, while those for relocatable code are found in the ‘Object’ directory.

We will be developing relocatable code, so you need to copy the appropriate template, such as ‘10F100TMPO.ASM’ or ‘12F508TMPO.ASM’, from the ‘mpasmx\templates\Object’ directory into the project folder created above, and give it a more meaningful name, such as ‘BA_L1-Turn_on_LED-10F200.asm’.

For example, with the names and paths given in the illustration for step 6 of the New Project wizard above, you would copy:

```
C:\Program Files\Microchip\MPLABX\mpasm\templates\Object\10F200TMPO.ASM
```

to

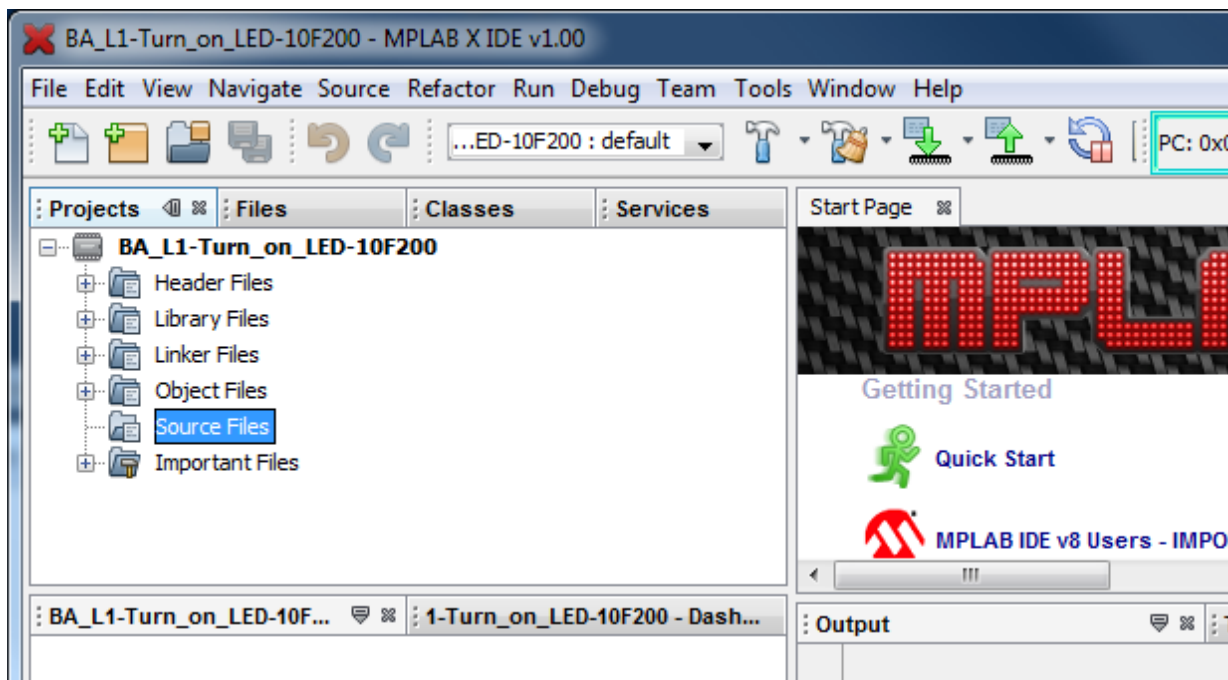
```
C:\...\Baseline\1 – Light an LED\BA_L1-Turn_on_LED-10F200\ BA_L1-Turn_on_LED-10F200.asm
```

Of course, you’ll put your project somewhere of your own choosing, and your MPASM installation may be somewhere else (especially if you are using Linux or a Mac), and if you’re using a PIC12F508 you’d substitute ‘12F508’ instead of ‘10F200’, but this should give you an idea of what’s needed.

Note that, unlike MPLAB 8, this copy step cannot be done from within the MPLAB X IDE. You need to use your operating system (Windows, Linux or Mac) to copy and rename the template file.

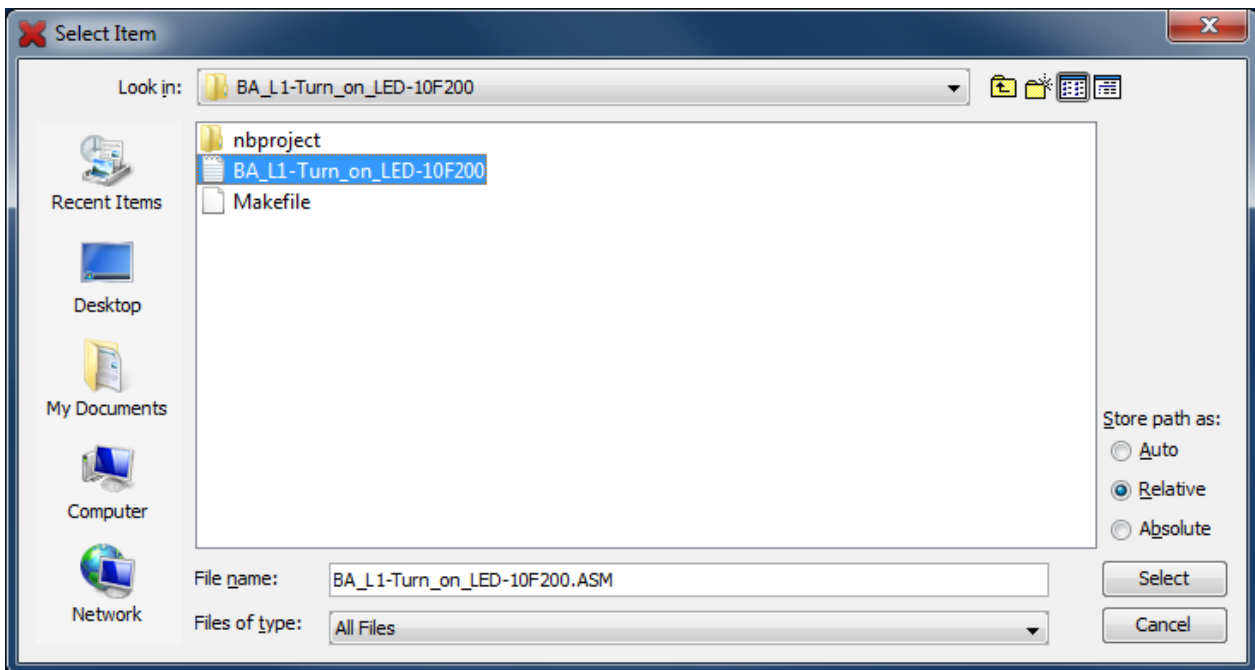
Now we need to tell MPASM X that this file is part of the project.

One way to do this is to right-click ‘Source Files’ in the project tree within the Projects window, and select ‘Add Existing Item...’:



⁹ On a 64-bit version of Windows, the MPASM folder will be under ‘Program Files (x86)’

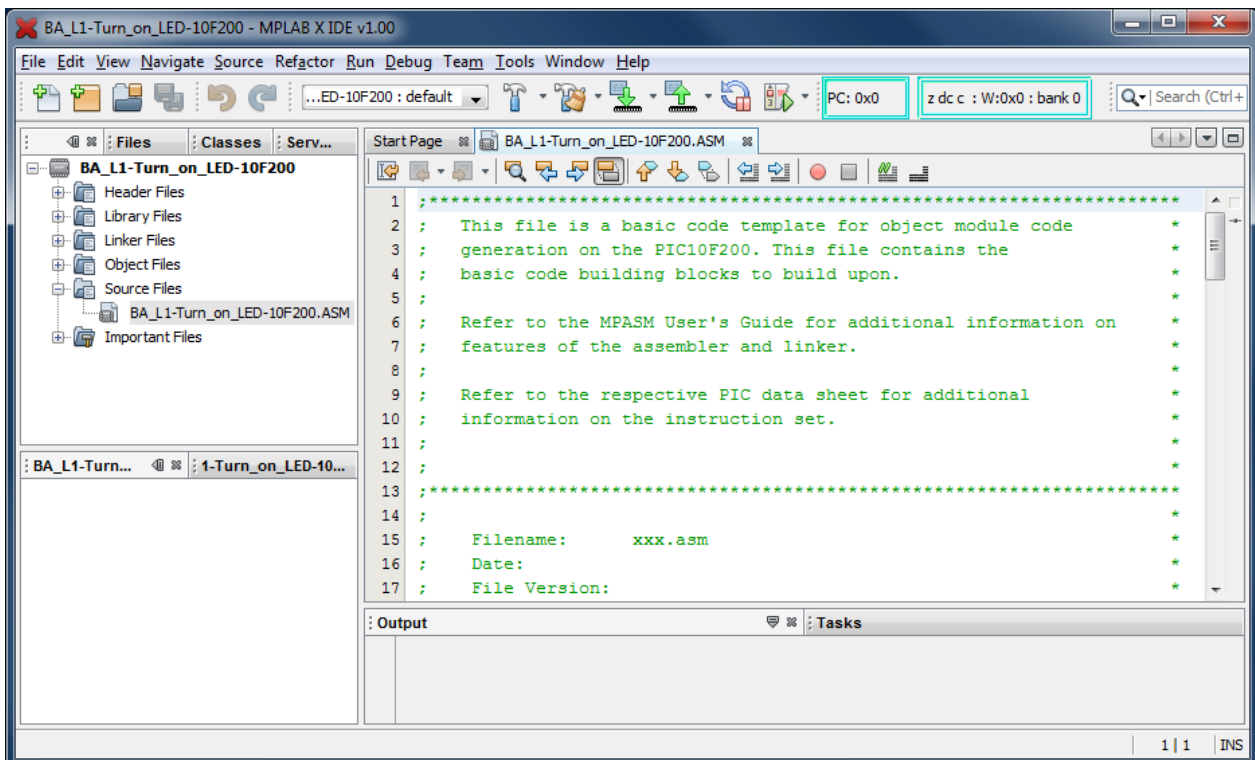
You can then select the template file that you copied (and renamed) into the project directory:



As with MPLAB 8, you need to tell MPLAB X whether the files you add are *relative* (they move with the project, if it is ever moved or copied) or *absolute* (remain in a fixed location). If you choose 'Auto', MPLAB X will guess. But since we know that this file is specific to your project, and should always move with it, you should select 'Relative' here.

Your .asm source file will now appear under 'Source Files' in the project tree.

If you double-click it, a text editor window will open, and you can finally start working on your code!



As in MPLAB 8, the text editor is aware of PIC assembler (MPASM) syntax and will colour-code the text. If you select ‘Options’ under the ‘Tools’ menu, you can set editor properties, such as tab size, but you’ll find that the defaults are quite usable to start with.

Template Code

The first section of the template is a series of blocks of comments.

MPASM comments begin with a ‘;’. They can start anywhere on a line. Anything after a ‘;’ is ignored by the assembler.

The template begins with some general instructions telling us that “this is a template” and “refer to the data sheet”. We already know all that, so the first block of comments can be deleted.

The following comment blocks illustrate the sort of information you should include at the start of each source file: what it’s called, modification date and version, who wrote it, and a general description of what it does. There’s also a “Files required” section. This is useful in larger projects, where your code may rely on other modules; you can list any dependencies here. It is also a good idea to include information on what processor this code is written for; useful if you move it to a different PIC later. You should also document what each pin is used for. It’s common, when working on a project, to change the pin assignments – often to simplify the circuit layout. Clearly documenting the pin assignments helps to avoid making mistakes when they are changed!

For example:

```
;*****
;
;  Filename:      BA_L1-Turn_on_LED-10F200.asm
;  Date:         2/1/12
;  File Version:  0.1
;
;  Author:       David Meiklejohn
;  Company:     Gooligum Electronics
;
;*****
;
;  Architecture: Baseline PIC
;  Processor:    10F200
;
;*****
;
;  Files required: none
;
;*****
;
;  Description:   Lesson 1, example 1
;
;  Turns on LED. LED remains on until power is removed.
;
;*****
;
;  Pin assignments:
;    GP1 = indicator LED
;
;*****
```

Note that the file version is ‘0.1’. I don’t call anything ‘version 1.0’ until it works; when I first start development I use ‘0.1’. You can use whatever scheme makes sense to you, as long as you’re consistent.

Next in the template, we find:

```
list      p=10F200          ; list directive to define processor
#include <p10F200.inc>      ; processor specific variable definitions
```

or, if you are using the PIC12F508, you would have:

```
list      p=12F508          ; list directive to define processor
#include <p12F508.inc>      ; processor specific variable definitions
```

The first line tells the assembler which processor to assemble for. It's not strictly necessary, as it is set in MPLAB (configured when you selected the device in the project wizard). MPLAB 8 displays the processor it's configured for at the bottom of the IDE window; see the screen shots above.

Nevertheless, you should always use the `list` directive at the start of your assembler source file. If you rely only on the setting in MPLAB, mistakes can easily happen, and you'll end up with unusable code, assembled for the wrong processor. If there is a mismatch between the `list` directive and MPLAB's setting, MPLAB will warn you when you go to assemble, and you can catch and correct the problem.

The next line uses the `#include` directive which causes an *include file* ('p10F200.inc' or 'p12F508.inc', located in the MPASM install directory) to be read by the assembler. This file sets up aliases for all the features of the processor, so that we can refer to registers etc. by name (e.g. 'GPIO') instead of numbers. [Lesson 6](#) explains how this is done; for now we'll simply use these pre-defined names, or *labels*.

These two things – the `list` directive and the include file – are specific to the processor. If you remember that, it's easy to move code to other PICs later.

Next we have, in the 10F200 template:

```
__CONFIG  _MCLRE_ON & _CP_OFF & _WDT_OFF
```

This sets the processor configuration. The 10F200 has a number of options that are set by setting various bits in a "configuration word" that sits outside the normal address space. The `__CONFIG` directive is used to set these bits as needed. We'll examine these in greater detail in later lessons, but briefly the options being set here are:

- `_MCLRE_ON`
Enables the external reset, or "master clear" ($\overline{\text{MCLR}}$) signal.

If enabled, the processor will be reset if pin 8 is pulled low. If disabled, pin 8 can be used as an input: GP3. That's why, on the circuit diagram, pin 8 is labelled "GP3/MCLR"; it can be either an input pin or an external reset, depending on the setting of this configuration bit.

The Gooligum training board includes a pushbutton which will pull pin 8 low when pressed, resetting the PIC if external reset is enabled. The PICkit 2 and PICkit 3 are also able to pull the reset line low, allowing MPLAB to control $\overline{\text{MCLR}}$ (if enabled) – useful for starting and stopping your program.

So unless you need to use every pin for I/O, it's a good idea to enable external reset by including '`_MCLRE_ON`' in the `__CONFIG` directive.

- `_CP_OFF`
Turns off code protection.

When your code is in production and you're selling PIC-based products, you may not want competitors stealing your code. If you use `_CP_ON` instead, your code will be protected, meaning that if someone tries to use a PIC programmer to read it, all they will see are zeros.

Since we're not designing anything for sale, we'll make our lives easier by leaving code protection turned off.

- `_WDT_OFF`

Disables the watchdog timer.

This is a way of automatically restarting a crashed program; if the program is running properly, it continually resets the watchdog timer. If the timer is allowed to expire, the program isn't doing what it should, so the chip is reset and the crashed program restarted – see [lesson 7](#).

The watchdog timer is very useful in production systems, but a nuisance when prototyping, so we'll leave it disabled.

The 12F508 template has a very similar `__CONFIG` directive:

```
__CONFIG _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC
```

The first three options are the same as before, but note that `MCLR` is on pin 4 on the 12F508, instead of pin 8. As on the 10F200, `MCLR` is shared with `GP3`. The LPC Demo Board also includes a pushbutton which will pull `GP3/MCLR` (pin 4) low when pressed. Unless we want to use the pushbutton on the LPC Demo Board as an input, it's best to leave it as a reset button, and enable external reset by specifying `'_MCLRE_ON'`.

The 12F508 `__CONFIG` directive also includes:

- `_IntRC_OSC`

This selects the internal RC oscillator as the clock source.

Every processor needs a clock – a regular source of cycles, used to trigger processor operations such as fetching the next program instruction.

Most modern PICs, including the 10F200 and 12F508, include an internal 'RC' oscillator, which can be used as the simplest possible clock source, since it's all on the chip! It's built from passive components – resistors and capacitors – hence the name RC.

The internal RC oscillator on the 10F200 and 12F508 runs at approximately 4 MHz. Program instructions are processed at one quarter this speed: 1 MHz, or 1 μ s per instruction.

Most PICs, including the 12F508, support a number of clock options, including more accurate crystal oscillators, as we'll see in [lesson 7](#), but the 10F200 does not; it only has the internal RC oscillator, which is why this wasn't part of the 10F200's `__CONFIG` directive.

To turn on an LED, we don't need accurate timing, so we'll stick with the internal RC oscillator, and include `'_IntRC_OSC'` in the 12F508's `__CONFIG` directive.

The comments following the `__CONFIG` directive in the template can be deleted, but it is a good idea to use comments to document the configuration settings.

For example:

```
;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, int RC clock
__CONFIG      _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC
```

The next piece of template code demonstrates how to define variables:

```
;***** VARIABLE DEFINITIONS
TEMP_VAR      UDATA
temp          RES      1                ;example variable definition
```

The `UDATA` directive tells the linker that this is the start of a *section* of uninitialised data. This is data memory space that is simply set aside for use later. The linker will decide where to place it in data memory. The label, such as `'TEMP_VAR'` here, is only needed if there is more than one `UDATA` section.

The `RES` directive is used to reserve a number of memory locations. Each location in data memory is 8 bits, or 1 byte, wide, so in this case, 1 byte is being reserved for a *variable* called `'temp'`. The address of the variable is assigned when the code is linked (after assembly), and the program can refer to the variable by name (i.e. `temp`), without having to know what its address in data memory is.

We'll use variables in later tutorials, but since we don't need to store any data to simply turn on an LED, this section can be deleted.

So far, we haven't seen a single PIC instruction. It's only been assembler or linker directives. The next piece of the 10F200 template code introduces our first instruction:

```
;*****
RESET_VECTOR    CODE    0xFF        ; processor reset vector

; Internal RC calibration value is placed at location 0xFF by Microchip
; as a movlw k, where the k is a literal value.
```

The `CODE` directive is used to introduce a *section* of program code.

The `0xFF` after `CODE` is an address in hexadecimal (signified in MPASM by the `'0x'` prefix). Program memory on the 10F200 extends from `000h` to `0FFh`. This `CODE` directive is telling the linker to place the section of code that follows it at `0x0FF` – the very top of the 10F200's program memory.

But there *is* no code following this first `CODE` directive, so what's going on? Remember that the internal RC oscillator is not as accurate as a crystal. To compensate for that inherent inaccuracy, Microchip uses a calibration scheme. The speed of the internal RC oscillator can be varied over a small range by changing the value of the `OSCCAL` register (refer back to the register map). Microchip tests every 10F200 in the factory, and calculates the value which, if loaded into `OSCCAL`, will make the oscillator run as close as possible to 4 MHz. This calibration value is inserted into an instruction placed at the top of the program memory (`0x0FF`). The instruction placed there is:

```
    movlw k
```

'`k`' is the calibration value inserted in the factory.

'`movlw`' is our first PIC assembler instruction. It loads the `W` register with an 8-bit value (between 0 and 255), which may represent a number, character, or something else.

Microchip calls a value like this, that is embedded in an instruction, a *literal*. It refers to a load or store operation as a 'move' (even though nothing is moved; the source never changes).

So, '`movlw`' means "move literal to **W**".

When the 10F200 is powered on or reset, the first instruction it executes is this `movlw` instruction at address `0x0FF`. After executing this instruction, the `W` register will hold the factory-set calibration value. And after executing an instruction at `0x0FF`, there is no more program memory¹⁰, so the *program counter*, which points to the next instruction to be executed, "wraps around" to point at the start of memory: `0x000`.

`0x0FF` is the 10F200's true "reset vector" (where code starts running after a reset), but `0x000` is the effective reset vector, where you place the start of your own code.

Confused?

¹⁰ The 10F200 has only 256 words of program memory, from address `000h` to `0FFh`.

What it really boils down to is that, when the 10F200 starts, it picks up a calibration value stored at the top of program memory, and then starts executing code from the start of program memory. Since you should never overwrite the calibration value at the top of memory, the start of your code will always be placed at 0x000, and when your code starts, the *W* register will hold the oscillator calibration value.

This “RESET_VECTOR” code section, as presented in the template, is not really very useful (other than comments telling you what you should already know from reading the data sheet), because it doesn’t actually stop your program overwriting the calibration value. Sure, it’s unlikely that your program would completely fill available memory, but to be absolutely sure, we can use the `RES` directive to reserve the address at the top of program memory:

```
;***** RC CALIBRATION
RCCAL   CODE    0x0FF           ; processor reset vector
        res 1                ; holds internal RC cal value, as a movlw k
```

Now, even if our program grows to fill all available program memory, the linker won’t allow us to overwrite the calibration value, because it’s been reserved.

The corresponding code section for the 12F508 version is much the same:

```
;***** RC CALIBRATION
RCCAL   CODE    0x1FF           ; processor reset vector
        res 1                ; holds internal RC cal value, as a movlw k
```

The only difference is that, because the 12F508 has 512 words of program memory, extending from address 000h to 1FFh, its calibration instruction is located at 0x1FF.

You can choose to use the oscillator calibration value, or simply ignore it. But if you’re using the internal RC oscillator, you should immediately copy this value to the `OSCCAL` register, to calibrate the oscillator with the factory setting, and that’s what the next piece of code from the template does:

```
MAIN    CODE    0x000
        movwf   OSCCAL           ; update register with factory cal value
```

This `CODE` directive tells the linker to place the following section of code at 0x000 – the effective reset vector.

The ‘`movwf`’ instruction copies (Microchip would say “moves”) the contents of the *W* register into the specified register – “**move W to file register**”.

In this case, *W* holds the factory calibration value, so this instruction writes that calibration value into the `OSCCAL` register.

Changing the labels and comments a little (for consistency with later lessons), we have:

```
;***** RESET VECTOR *****
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL           ; apply internal RC factory calibration
```

At this point, all the preliminaries are out of the way. The processor has been specified, the configuration set, the oscillator calibration value updated, and program counter pointing at the right location to run user code.

If this seems complicated (and unfortunately, it is!), at least the worst is over. We can finally start the main part of the program, and focus on actually implementing the application.

The final piece of template code is simply an example showing where and how to place your code:

```
start
    nop                ; example code
    movlw    0xFF      ; example code
    movwf   temp      ; example code

; remaining code goes here

                END                ; directive 'end of program'
```

‘start’ is an example of a program label, used in loops, branches and subroutines. It’s not necessary to label the start of your code ‘start’. But it does make it easier to follow the code.

‘nop’ is a “no operation” instruction; it does nothing other than waste an instruction cycle – something you might want to do as part of a delay loop (we’ll look at examples in the [next lesson](#)).

‘movlw’ and ‘movwf’ we’ve seen before.

‘END’ is an assembler directive, marking the end of the program source. The assembler will ignore any text after the ‘END’ directive – so it really should go right at the end!

Of course, we need to replace these example instructions with our own. This is where we place the code to turn on the LED!

Turning on the LED

To turn on the LED on GP1, we need to do two things:

- Configure GP1 as an output
- Set GP1 to output a high voltage

We could leave the other pins configured as inputs, or set them to output a low. Since, in this circuit, they are not connected to anything, it doesn’t really matter. But for the sake of this exercise, we’ll configure them as inputs.

When a baseline PIC is powered on, all pins are configured by default as inputs, and the content of the port register, GPIO, is undefined.

To configure GP1 as an output, we have to write a ‘0’ to bit 1 of the TRIS register. This is done by:

```
    movlw    b'111101'    ; configure GP1 (only) as an output
    tris    GPIO
```

The ‘tris’ instruction stores the contents of W into a **TRIS** register.

Although there is only one TRIS register on the 10F200 or 12F508, it is still necessary to specify ‘GPIO’ (or equivalently the number 6, but that would be harder to follow) as the operand.

Note that to specify a binary number in MPASM, the syntax b‘*binary digits*’ is used, as shown.

To set the GP1 output to ‘high’, we have to set bit 1 of GPIO to ‘1’. This can be done by:

```
    movlw    b'000010'    ; set GP1 high
    movwf   GPIO
```

As the other pins are all inputs, it doesn't matter what they are set to.

Note again that, to place a value into a register, you first have to load it into W. You'll find that this sort of load/store process is common in PIC programming.

Finally, if we leave it there, when the program gets to the end of this code, it will restart. So we need to get the PIC to just sit doing nothing, indefinitely, with the LED still turned on, until it is powered off.

What we need is an "infinite loop", where the program does nothing but loop back on itself, indefinitely. Such a loop could be written as:

```
here    goto    here
```

'here' is a label representing the address of the `goto` instruction.

'goto' is an unconditional branch instruction. It tells the PIC to **go to** a specified program address.

This code will simply go back to itself, always. It's an infinite, do-nothing, loop.

A shorthand way of writing the same thing, that doesn't need a unique label, is:

```
goto    $                ; loop forever
```

'\$' is an assembler symbol meaning the current program address.

So this line will always loop back on itself.

This little program, although small, has a structure common to most PIC programs: an initialisation section, where the I/O pins and other facilities are configured and initialised, followed by a "main loop", which repeats forever. Although we'll add to it in future lessons, we'll always keep this basic structure of initialisation code followed by a main loop.

Complete program

Putting together all the above, here's the complete assembler source needed for turning on an LED, for the PIC10F200:

```

;*****
;
;   Description:      Lesson 1, example 1
;
;   Turns on LED.   LED remains on until power is removed.
;
;*****
;
;   Pin assignments:
;       GP1 = indicator LED
;
;*****

list          p=10F200
#include      <p10F200.inc>

;***** CONFIGURATION
;           ; ext reset, no code protect, no watchdog
__CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF

```



```

;***** RC CALIBRATION
RCCAL   CODE    0x0FF           ; processor reset vector
        res 1                   ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET   CODE    0x000           ; effective reset vector
        movwf  OSCCAL           ; apply internal RC factory calibration

;***** MAIN PROGRAM *****

;***** Initialisation
start
        movlw  b'111101'       ; configure GP1 (only) as an output
        tris   GPIO
        movlw  b'000010'       ; set GP1 high
        movwf  GPIO

;***** Main loop
        goto   $               ; loop forever

        END

```

The 12F508 version is very similar, with changes to the `list`, `#include`, `__CONFIG` and `RCCAL CODE` directives, as noted earlier.

That's it! Not a lot of code, really...

Building the Application and Programming the PIC

Now that we have the complete assembler source, we can build the final application code and program it into the PIC.

This is done in two steps:

- Build the project
- Use a programmer to load the program code into the PIC

The first step, building the project, involves assembling the source files¹¹ to create object files, and linking these object files, to build the executable code. Normally this is transparent; MPLAB does all of this for you in a single operation. The fact that, behind the scenes, there are multiple steps only becomes important when you start working with projects that consist of multiple source files or libraries of pre-assembled routines.

A PIC programmer, such as the PICkit 2 or PICkit 3, is then used to upload the executable code into the PIC. Although a separate application is sometimes used for this “programming” process, it’s convenient when developing code to do the programming step from within MPLAB, which is what we’ll look at here.

Although the concepts are the same, the details of building your project and programming the PIC depend on the IDE you are using, so we’ll look at how it’s done in both MPLAB 8 and MPLAB X.

¹¹ Although there is only one source file in this simple example, larger projects often consist of multiple files; we’ll see an example in [lesson 3](#).

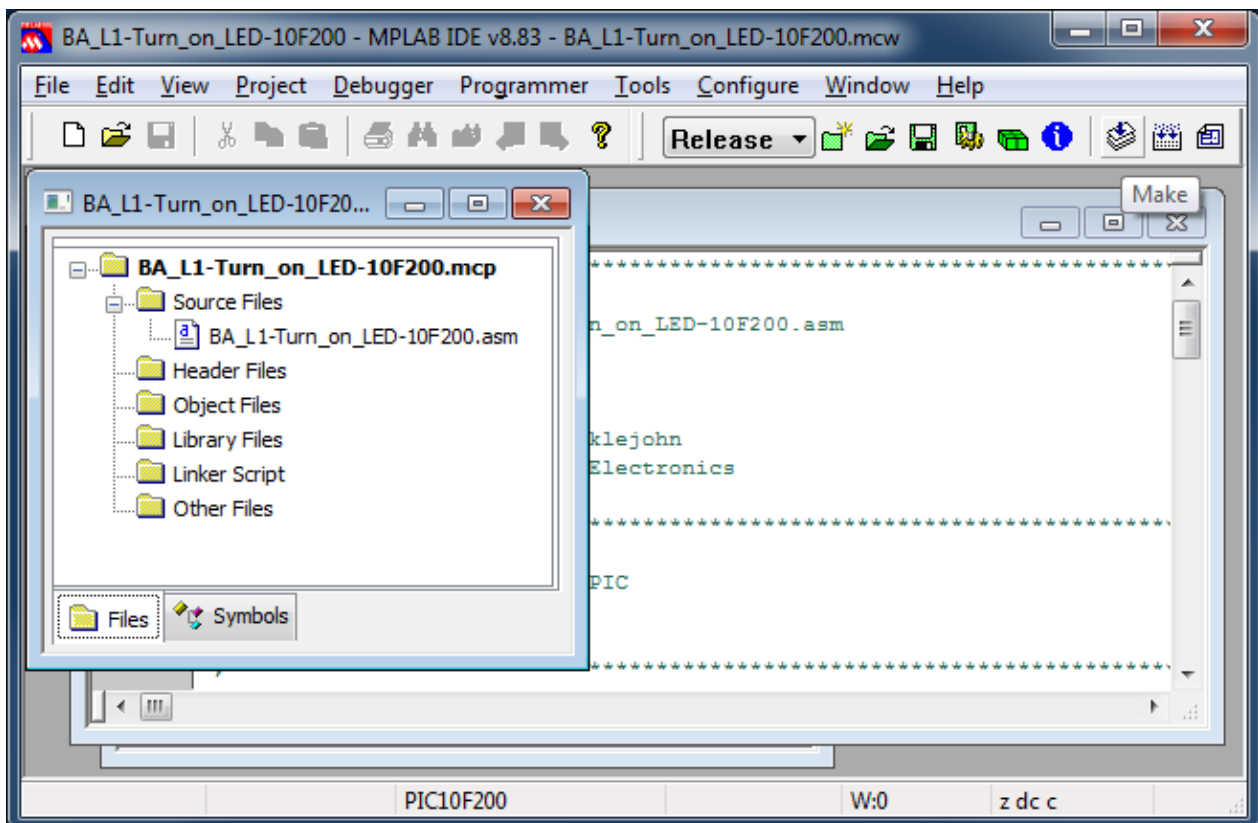
MPLAB 8.xx

Building the project

When you build a project using MPLAB 8, it needs to know whether you will be using a hardware debugger to debug your code.

We won't be debugging, so select "Release" under the "Project → Build Configuration" menu item, or, more conveniently, select "Release" from the drop-down menu in the toolbar.

To build the project, select the "Project → Make" menu item, press F10, or click on the "Make" button in the toolbar:



"Make" will assemble any source files which need assembling (ones which have changed since the last time the project was built), then link them together.

The other option is "Project → Build All" (Ctrl+F10), which assembles all the source files, regardless of whether they have changed (are "out of date") or not.

For a small, single-file project like this, "Make" and "Build All" have the same effect; you can use either. In fact, the only reason to use "Make" is that, in a large project, it saves time to not have to re-assemble everything each time a single change is made.

When you build the project (run "Make"), you'll see text in the Output window showing the code being assembled, similar to this:

```
Make: The target "C:\Work\Gooligum\Tutorials\Series 2\Baseline\1 - Light an LED\BA_L1-Turn_on_LED-10F200.o" is out of date.
Executing: "C:\Program Files\Microchip\MPASM Suite\MPASMWIN.exe" /q /p10F200 "BA_L1-Turn_on_LED-10F200.asm" /I"BA_L1-Turn_on_LED-10F200.lst" /e"BA_L1-Turn_on_LED-10F200.err" /o"BA_L1-Turn_on_LED-10F200.o"
```

If you see any errors or warnings, you probably have a syntax error somewhere, so check your code against the listing above.

The next several lines show the code being linked:

```

Make: The target "C:\Work\Gooligum\Tutorials\Series 2\Baseline\1 - Light an LED\BA_L1-Turn_on_LED-10F200.cof" is out of date.
Executing: "C:\Program Files\Microchip\MPASM Suite\mplink.exe" /p10F200 "BA_L1-Turn_on_LED-10F200.o" /z__MPLAB_BUILD=1 /o"BA_L1-Turn_on_LED-10F200.cof" /M"BA_L1-Turn_on_LED-10F200.map" /W
MPLINK 4.41, Linker
Device Database Version 1.5
Copyright (c) 1998-2011 Microchip Technology Inc.
Errors : 0

MP2HEX 4.41, COFF to HEX File Converter
Copyright (c) 1998-2011 Microchip Technology Inc.
Errors : 0

Loaded C:\Work\Gooligum\Tutorials\Series 2\Baseline\1 - Light an LED\BA_L1-Turn_on_LED-10F200.cof.

```

The linker, “MPLINK”, creates a “COFF” object module file, which is converted into a “.hex” hex file, which contains the actual machine codes to be loaded into the PIC.

In addition to the hex file, other outputs of the build process include a “.lst” list file which allows you to see how MPASM assembled the code and the values assigned to symbols, and a “.map” map file, which shows how MPLINK laid out the data and code segments in the PIC’s memory.

Programming the PIC

The final step is to upload the final assembled and linked code into the PIC.

First, ensure that you have connected you PICKit 2 or PICKit 3 programmer to your Gooligum training board or Microchip LPC Demo Board, with the PIC correctly installed in the appropriate IC socket¹², and that the programmer is plugged into your PC.

You can now select your programmer from the “Programmer → Select Programmer” menu item.

If you are using a PICKit 2, you should see messages in a “PICKit 2” tab in the Output window, similar to these:

```

Initializing PICKit 2 version 0.0.3.63
Found PICKit 2 - Operating System Version 2.32.0
Target power not detected - Powering from PICKit 2 ( 5.00V)
PICKit 2 Ready

```

If the operating system in the PICKit 2 is out of date, you will see some additional messages while it is updated.

If you don’t get the “Found PICKit 2” and “PICKit 2 Ready” messages, you have a problem somewhere and should check that everything is plugged in correctly.

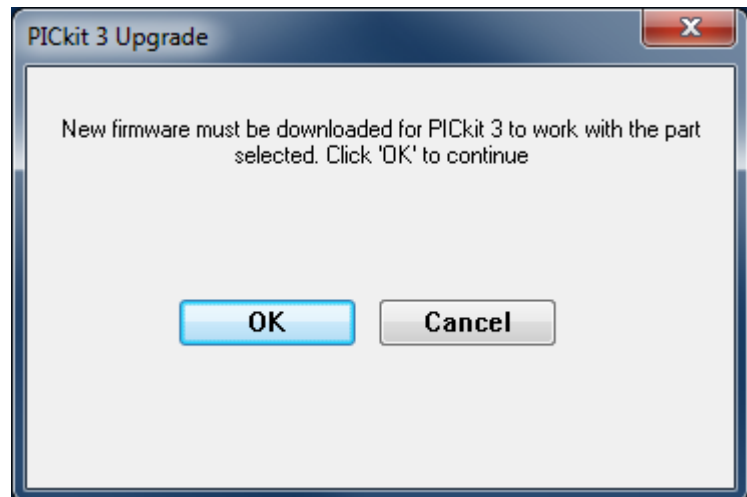
¹² Or, in general, that the PIC you wish to program is connected to whichever programmer or debugger you are using, whether it’s in a demo/development/training board, a production board, or a standalone programmer.

If you are using a PICkit 3, you may see a message telling you new firmware must be downloaded.

This is because, unlike the PICkit 2, the PICkit 3 uses different firmware to support each PIC device family, such as baseline or mid-range.

You'll only see this prompt when you change to a new device family.

If you do see it, just click 'OK'.

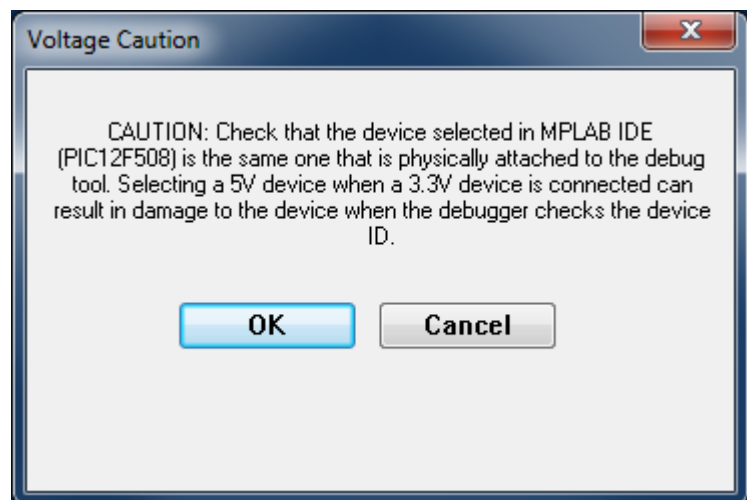


You may also see a voltage caution warning, as shown on the right.

This is because many newer PIC devices, including those with 'LF' in their part number, do not support 5 V operation, and can be damaged if connected to a 5 V supply.

The PICkit 3 will supply 5 V if you select a 5 V device, such as the 10F200 or 12F508.

We are using a 5 V device, so you can click 'OK'.

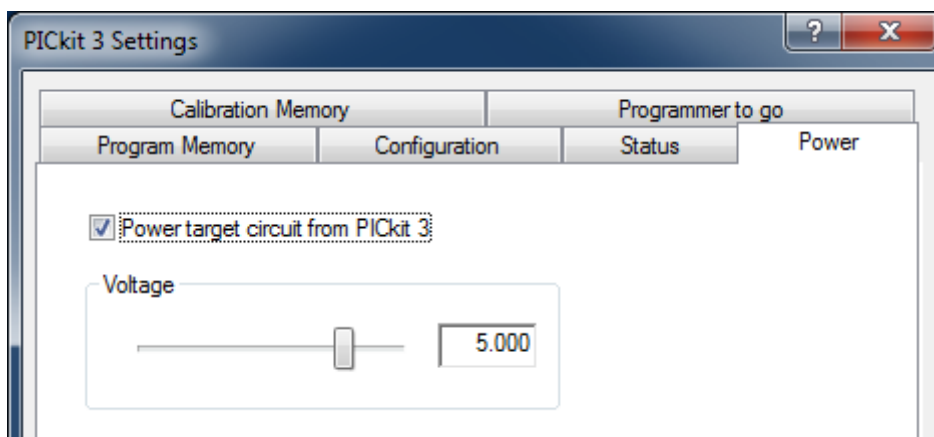


You may now see an error message in the output window, such as:

PK3Err0045: You must connect to a target device to use PICkit 3.

This is because, despite the previous warning about power, the PICkit 3 will not actually supply power to our circuit (necessary to program the PIC, unless you have separate power supply), until you tell it to do so.

Select the "Programmer → Settings" menu item. This will open the PICkit 3 Settings window. In the "Power" tab, select "Power target circuit from PICkit 3", as shown:



You can leave the voltage set to 5.0 V, and then click 'OK'.

You'll get the voltage caution again, so click 'OK' again.

If you now choose the “Programmer → Reconnect” menu item, you should see messages in a “PICkit 3” tab in the Output window (after yet another voltage caution), similar to these:

```
PICkit 3 detected
Connecting to PICkit 3...
Firmware Suite Version..... 01.26.92
Firmware type.....Baseline
PICkit 3 Connected.
```

It does seem that the PICkit 3 is a little fiddlier to work with, than the PICkit 2...

After you select your programmer, an additional toolbar will appear.

For the PICkit 2, it looks like:



For the PICkit 3, we have:



As you can see, they are very similar.

The first icon (on the left) is used to initiate programming. When you click on it, you should see messages like:

```
Programming Target (5/01/2012 12:41:49 PM)
Erasing Target
Programming Program Memory (0x0 - 0x7)
Verifying Program Memory (0x0 - 0x7)
Programming Configuration Memory
Verifying Configuration Memory
PICkit 2 Ready
```


Or, if you are using a PICkit 3, simply:

```
Programming...
Programming/Verify complete
```

Your PIC is now programmed!

If you are using a PICkit 3, the LED on GP1 should immediately light up.

If you have a PICkit 2, you won't see anything yet. That is because, by default, the PICkit 2 holds the $\overline{\text{MCLR}}$ line low after programming. Since we have used the `_MCLRE_ON` configuration option, enabling external reset, the PIC is held in reset and the program will not run. If we had not configured external resets, the LED would have lit as soon as the PIC was programmed.

To allow the program to run, click on the  icon, or select the “Programmer → Release from Reset” menu item.

The LED should now light!

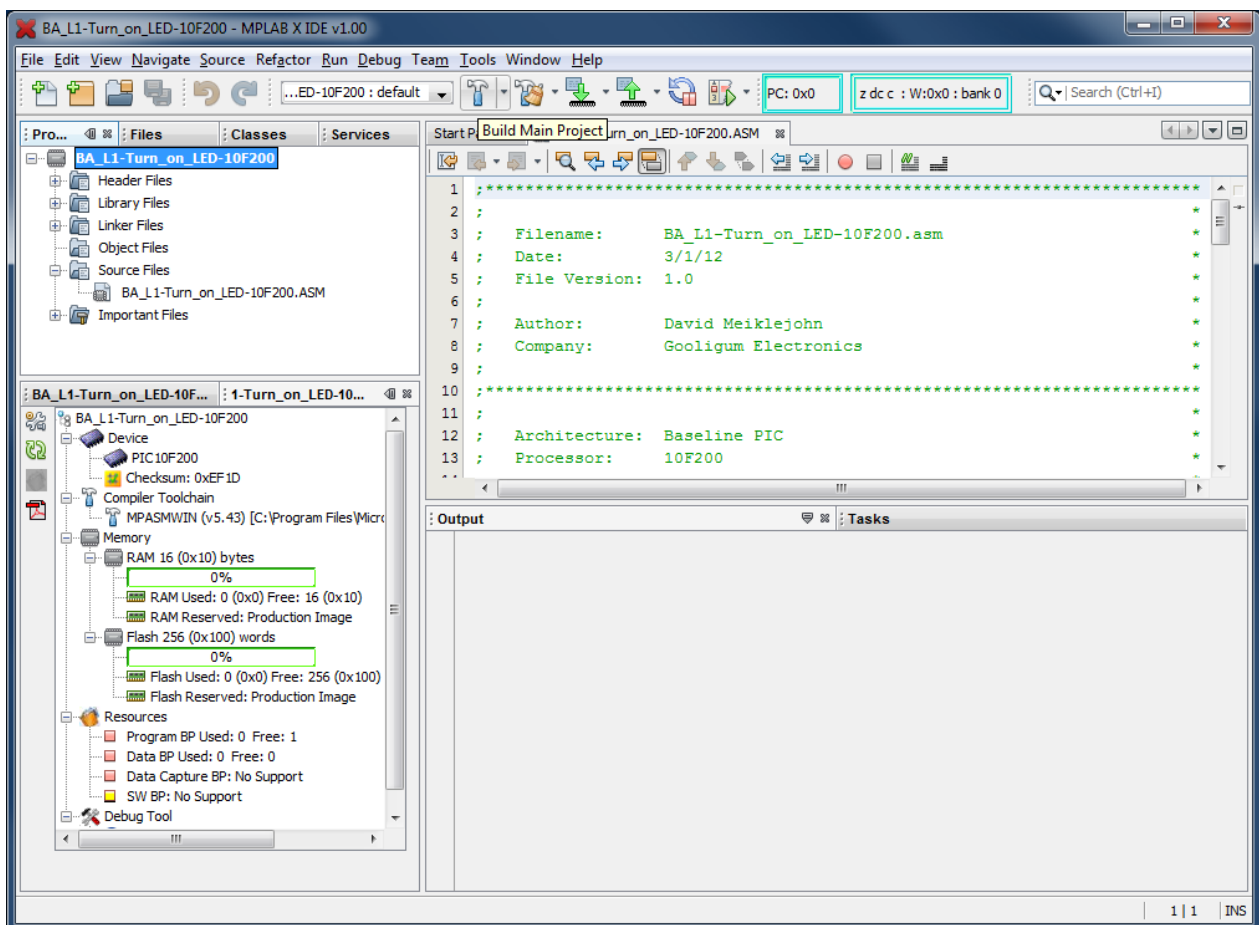
MPLAB X

Building the project

Before you build your project using MPLAB X, you should first ensure that it is the “main” project. It should be highlighted in bold in the Projects window.

To set the project you want to work on (and build) as the main project, you should right-click it and select “Set as Main Project”. If you happen to have more than one project in your project window, you can by removing any project you are not actively working on (to reduce the chance of confusion) from the Projects window, by right-clicking it and selecting “Close”.

To build the project, right-click it in the Projects window and select “Build”, or select the “Run → Build Main Project” menu item, or simply click on the “Build Main Project” button (looks like a hammer) in the toolbar:



This will assemble any source files which have changed since the project was last built, and link them.

An alternative is “Clean and Build”, which removes any assembled (object) files and then re-assembles all files, regardless of whether they have been changed. This action is available by right-clicking in the Projects window, or under the “Run” menu, or by clicking on the “Clean and Build Main Project” button (looks like a hammer with a brush) in the toolbar.

When you build the project, you’ll see messages in the Output window, showing your source files being assembled and linked. At the end, you should see:

```
BUILD SUCCESSFUL (total time: 2s)
```

(of course, your total time will probably be different...)

If, instead, you see an error message, you'll need to check your code and your project configuration.

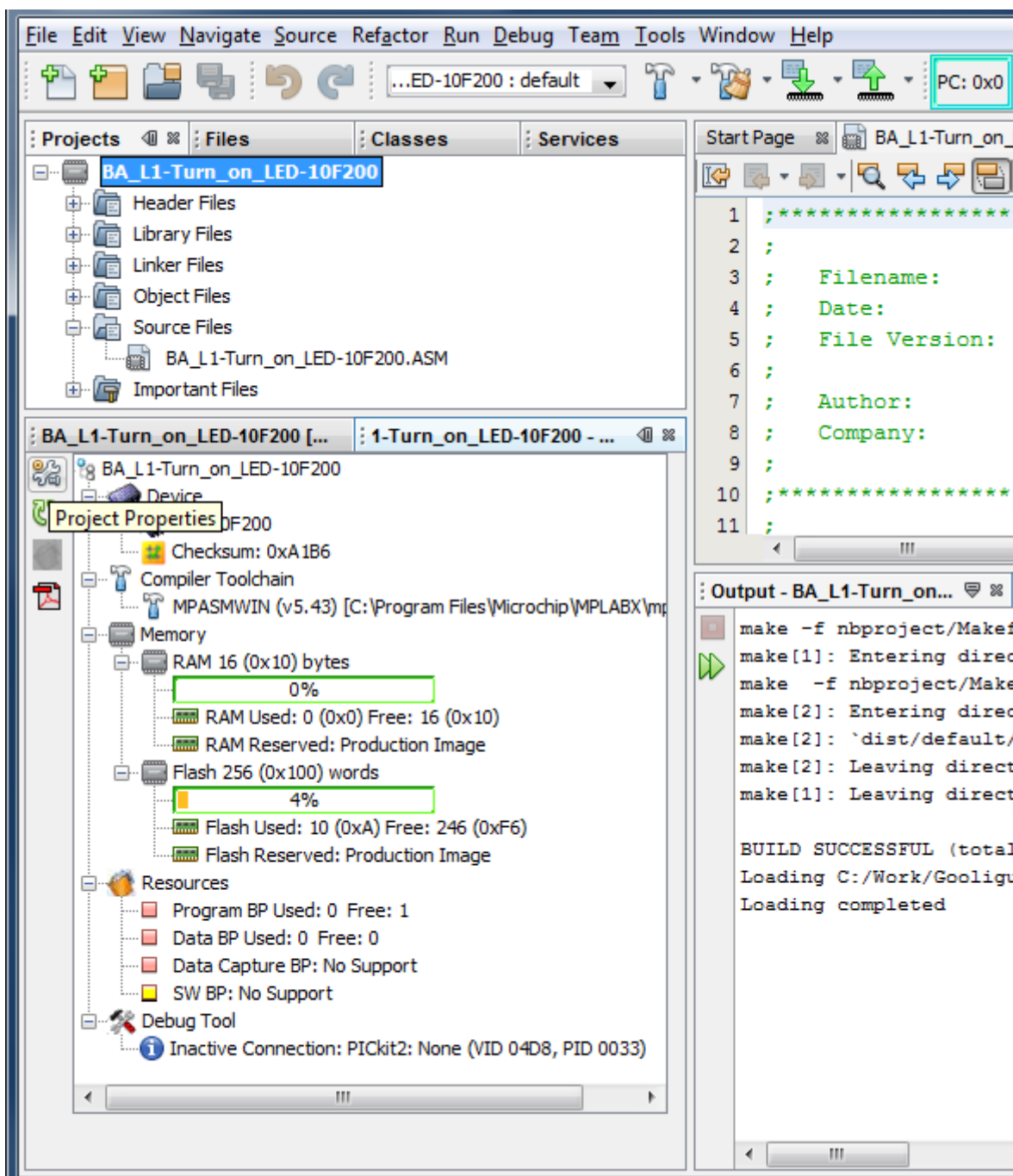
Programming the PIC

The final step is to upload the executable code into the PIC.

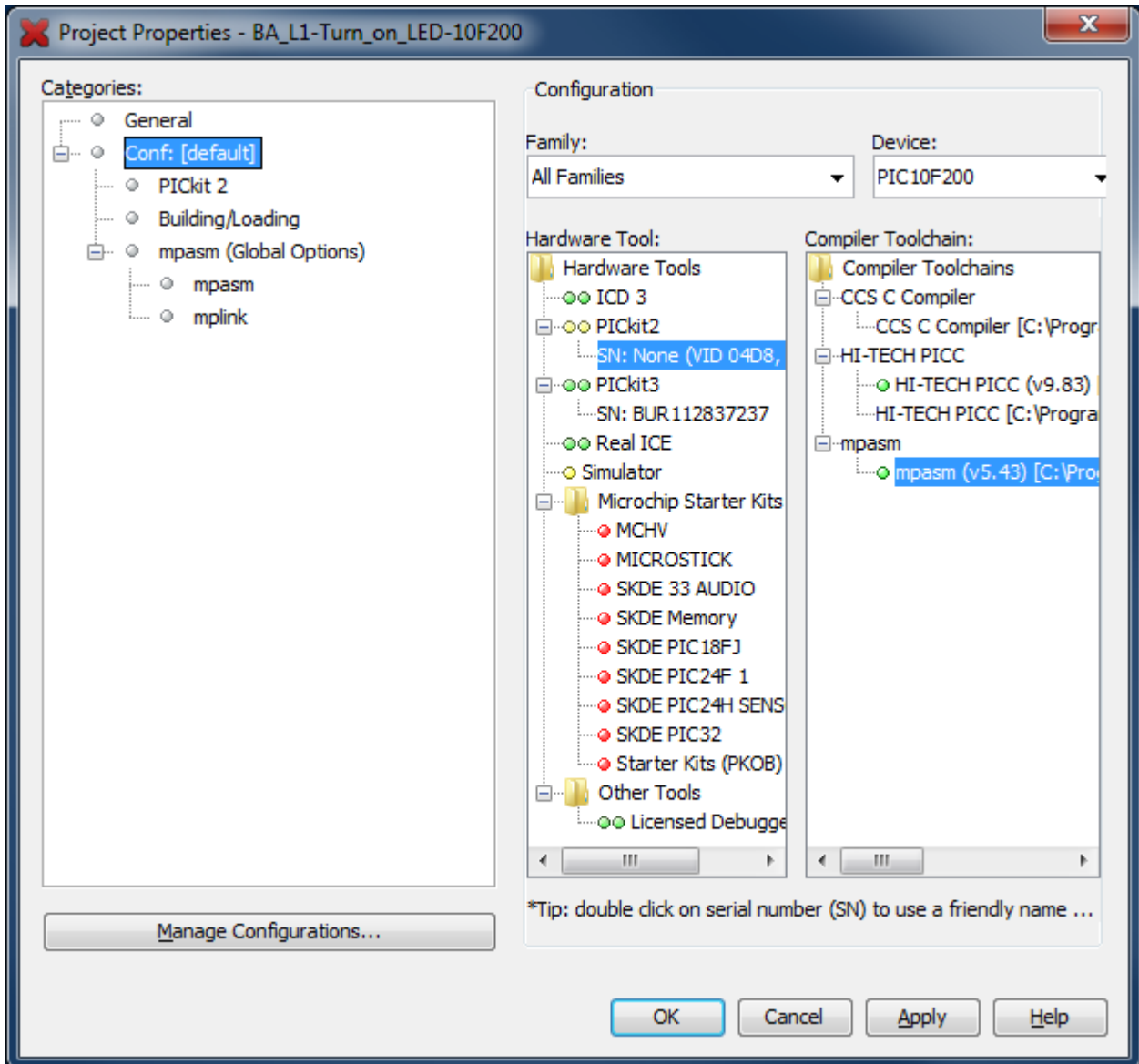
First, ensure that you have connected your PICkit 2 or PICkit 3 programmer to your Gooligum training board or Microchip LPC Demo Board, with the PIC correctly installed in the appropriate IC socket, and that the programmer is plugged into your PC.

If you have been following this lesson, you will have specified the programmer when you created your project (in step 4 of the wizard).

If you want to check that the correct programmer is selected, or if you want to change your tool selection, you can right-click your project in the Projects window and select "Properties", or simply click on the "Project Properties" button on the left side of the Project Dashboard:



This will open the project properties window, where you can verify or change your hardware tool (programmer) selection:



After closing the project properties window, you can now program the PIC.

You can do this by right-clicking your project in the Projects window, and select “Make and Program Device”. This will repeat the project build, which we did earlier, but because nothing has changed (we have not edited the code), the “make” command will see that there is nothing to do, and the assembler will not run.

Instead, you should see output like:

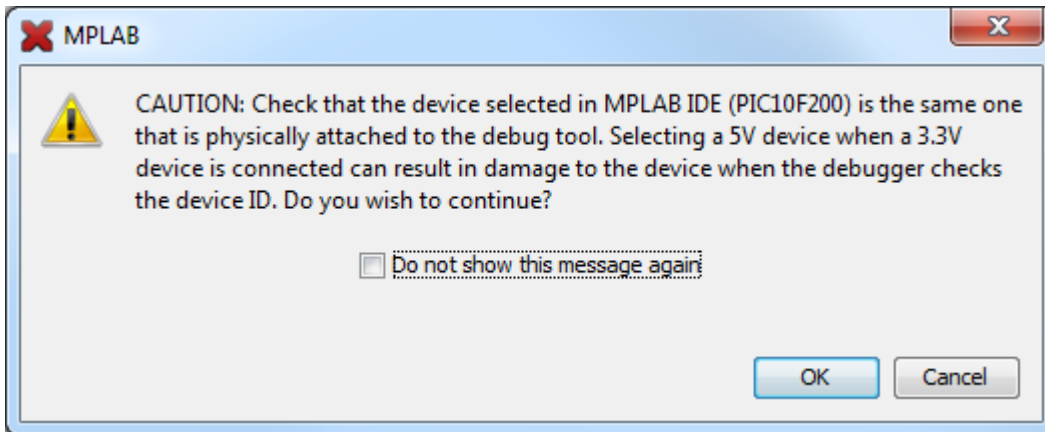
```
BUILD SUCCESSFUL (total time: 266ms)
Loading C:/Work/Gooligum/Tutorials/Series 2/Baseline/1 - Light an LED/BA_L1-Turn_on_LED-10F200/dist/default/production/BA_L1-Turn_on_LED-10F200.production.hex...
Loading completed
Connecting to programmer...
Programming target...
```

(the total time is much smaller than before, because no assembly had to be done).

If you are using a PICKit 3, MPLAB X will download new firmware into it, and you will see messages in the PICKit 3 output window like:

```
Downloading Firmware...
Downloading AP...
AP download complete
Firmware Suite Version.....01.27.20
Firmware type.....Baseline
```

You may also see a voltage caution warning, as shown below:



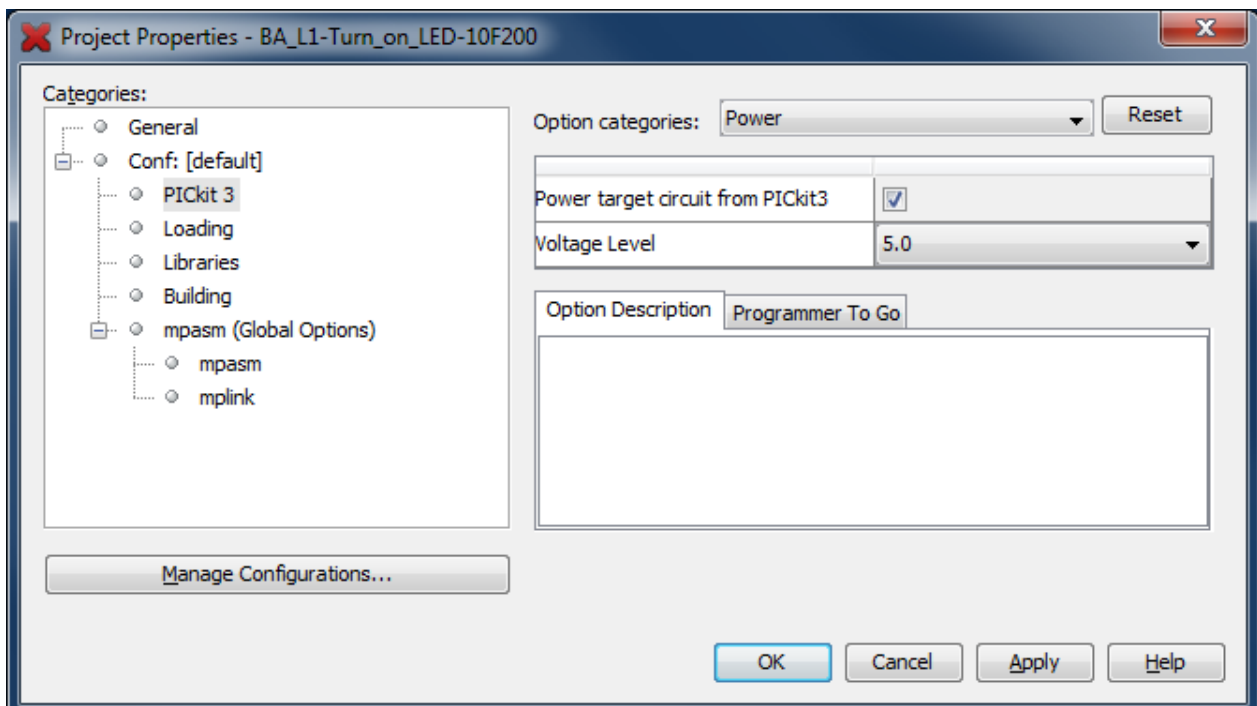
Since we are using a 5 V device, you can click 'OK'. And feel free to click "Do not show this message again", to avoid seeing this caution every time you program your PIC.

You may now see an error message in the PICKit 3 output window, stating:

```
Target device was not found. You must connect to a target device to use PICKit 3.
```

This happens if the PIC is unpowered, so we need to tell the PICKit 3 to supply power.

Open the project properties window (as on the previous page), select 'PICKit 3' in the categories tree, and choose the 'Power' option in the drop-down categories list:



Select “Power target circuit from PICKit3”, as shown. You can leave the voltage set to 5.0 V, and then click ‘OK’.

If you now perform “Make and Program Device” again, the programming should be successful and you should see, in the build output window, messages ending in:


Programming completed

Note that this action combines making (or building) the project, with programming the PIC. What were two steps in MPLAB 8 are combined into one step here. In fact, with MPLAB X, there is no straightforward way to simply program the PIC, without building your project as well.

This makes sense, because you will almost always want to program your PIC with the latest code. If you make a change in the editor, you want to program that change into the PIC. With MPLAB X, you can be sure that whatever code you see in your editor window is what will be programmed into the PIC.

But most times, you’ll want to go a step further, and run your program, after uploading it into the PIC, to see if it works. For that reason, MPLAB X makes it very easy to build your code, program it into your PIC, and then run it, all in a single operation.

There are a few ways to do this:


- Right-click your project in the Projects window, and select “Run”, or
- Select the “Run → Run Main Project” menu item, or
- Press ‘F6’, or
- Click on the “Make and Program Device” button in the toolbar: 


Whichever of these you choose, you should see output messages ending in:

Running target...

The LED on GP1 should now light.

Being able to build, program and run in a single step, by simply pressing ‘F6’ or clicking on the “Make and Program Device” button is very useful, but what if you don’t want to automatically run your code, immediately after programming?

If you want to avoid running your code, click on the “Hold in Reset” toolbar button () before programming. You can now program your PIC as above.


Your code won’t run until you click the reset toolbar button again, which now looks like  and is now tagged as “Release from Reset”.

Summary

The sections above, on building your project and programming the PIC, have made using MPLAB X seem much more complicated than it really is.

Certainly, there are a lot of options and ways of doing things, but in practice it’s very simple.

Most of the time, you will be working with a single project, and only one hardware tool, such as a programmer or debugger, which you will have selected when you first ran the New Project wizard.

In that case (and most times, it will be), just press 'F6' or click on  to build, program and run your code – all in a single, easy step.

That's all there is to it. Use the New Project wizard to create your project, add a template file to base your code on, use the editor to edit your code, and then press 'F6'.

Conclusion

For such a simple task as lighting an LED, this has been a very long lesson!

In summary, we:

- Introduced two baseline PICs:
 - 10F200
 - 12F508
- Showed how to configure and use the PIC's output pins
- Implemented an example circuit using two development boards:
 - Gooligum training and development board
 - Microchip Low Pin Count Demo Board
- Looked at Microchip's assembly template code and saw:
 - some PIC assembler directives
 - some PIC configuration options
 - our first few PIC instructions
- Modified it to create our (very simple!) PIC program
- Introduced two development environments:
 - MPLAB 8.xx
 - MPLAB X
- Showed how to use these development environments to:
 - Create a new project
 - Include existing template code
 - Modify that template code
 - Build the program
 - Program the PIC, using:
 - PICKit 2
 - PICKit 3
 - Run the program

That is a lot, to accomplish so little – although you can of course ignore the sections that aren't relevant to your environment. You should use MPLAB 8 if it's still available, supported, and works with your PC, in which case you can ignore the MPLAB X sections for now, and come back to them if you upgrade later.

If you have the Gooligum training board, you can ignore sections about the Microchip LPC Demo board. And if you're lucky enough to have a PICkit 2, you can ignore the sections about the PICkit 3.

Nevertheless, after all this, you have a solid base to build on. You have a working development environment. You can create projects, modify your code, load (program) your code into your PIC, and make it run.

Congratulations! You've taken your first step in PIC development!

That first step is the hardest. From this point, we build on what's come before.

In the [next lesson](#), we'll make the LED flash...

Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 2: Flash an LED

In [lesson 1](#) we lit a single LED connected to one of the pins of a PIC10F200 or PIC12F508.

Now we'll make it flash.

In doing this, we will learn about:

- Using loops to create delays
- Variables
- Using exclusive-or (xor) to flip bits
- The 'read-modify-write' problem

The development environments and microcontrollers used for this lesson are the same as those in lesson 1.

Again, it is assumed that you are using a Microchip PICkit 2 or PICkit 3 programmer and either the [Gooligum Baseline and Mid-Range PIC Training and Development Board](#) or Microchip's Low Pin Count (LPC) Demo Board, with Microchip's MPLAB 8 or MPLAB X integrated development environment. But it is of course possible to adapt these instructions to a different programmers and/or development boards.

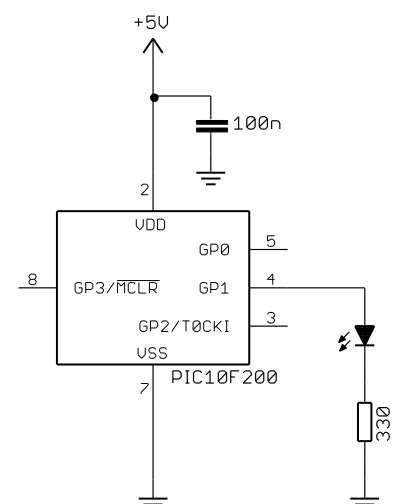
We will also assume that, if you have the Gooligum training board, you will continue to use the PIC10F200, and that if you have the Microchip LPC Demo Board, you will be using a PIC12F508 – both introduced in lesson 1.

Example Circuit

Here's the PIC10F200 version of the circuit again.

If you have the Gooligum training board, simply plug the PIC10F200 into the 8-pin IC socket marked '10F'.

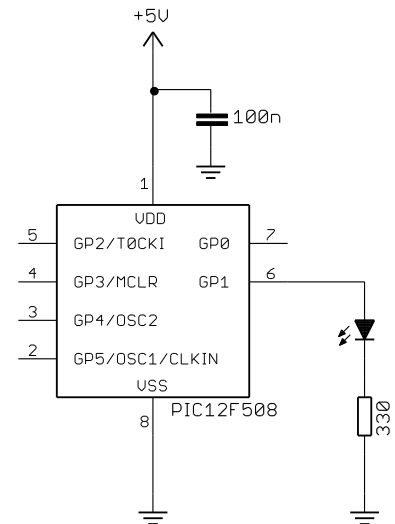
Connect a shunt across the jumper (JP12) on the LED labelled 'GP1', and ensure that every other jumper is disconnected.



Here's the corresponding PIC12F508 version.

You will need to use a PIC12F508 if you have Microchip's Low Pin Count Demo Board.

Refer back to [lesson 1](#) to see how to build this circuit, either by soldering a resistor, LED (and optional isolating jumper) to the demo board, or by making connections on the demo board's 14-pin header.



Creating a new project

It is a good idea, where practical, to base a new software project on work you've done before. In this case, it makes sense to build on the program from lesson 1 – we just have to add extra instructions to flash the LED.

How to create a new project, based on an existing one, depends on whether you're using MPLAB 8 or MPLAB X, so we'll take a look at both.

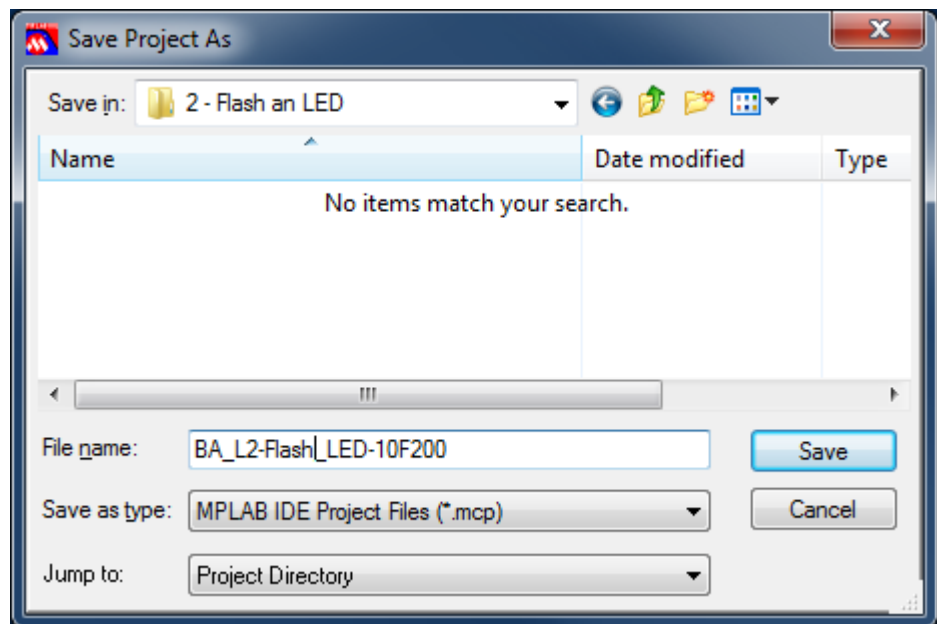
MPLAB 8.xx

There are a couple of ways to do this, but the following method works well.

First, open the project you created in lesson 1 in MPLAB 8. You can do this easily by double-clicking the '*.mcp' project file in your project folder.

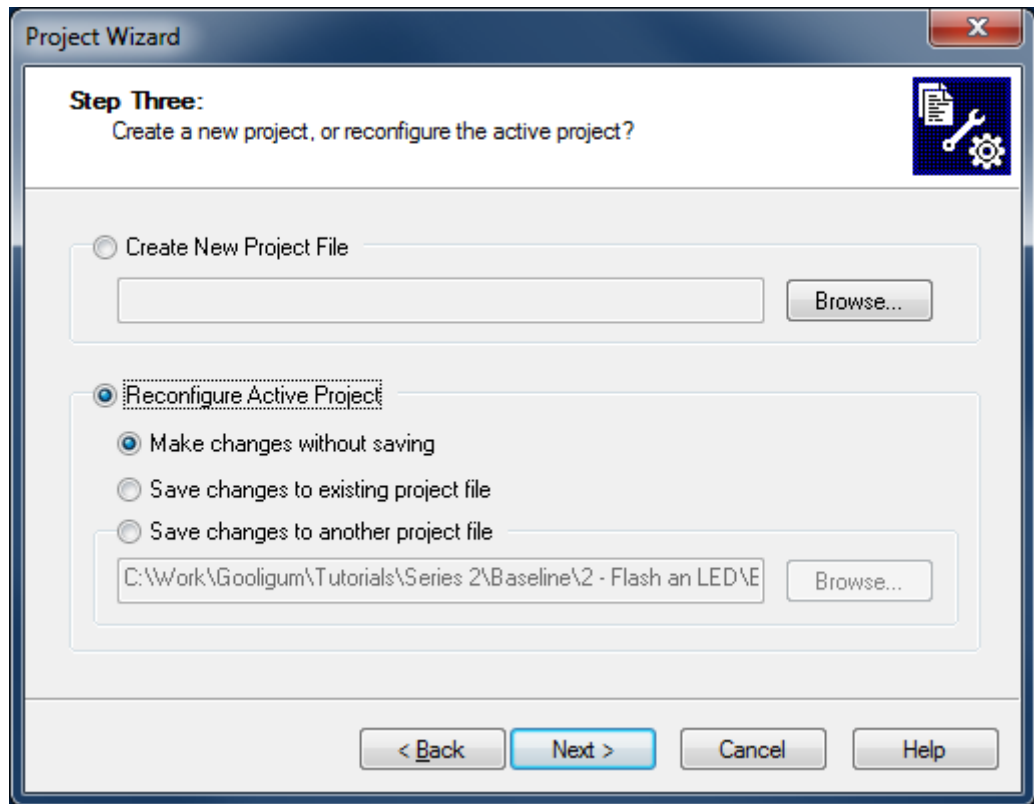
Now use the "Project → Save Project As..." menu item to save the project in a new folder, with a new name.

When a project is saved to a new location, all the files belonging to that project ("User" files, with relative paths) are copied to that location. You will find that in this case the '*.asm' source file from lesson 1 has been copied into your new folder.

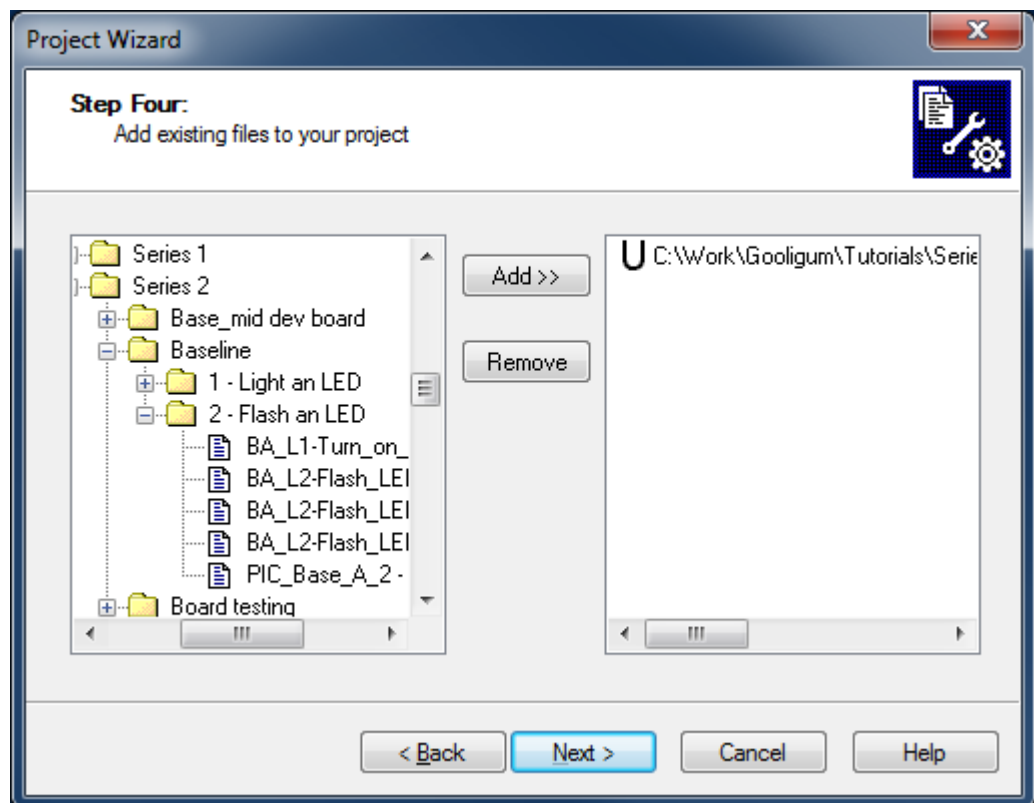


The next step is to use the project wizard (“Project → Project Wizard...”) to reconfigure the project, giving the source file a new name.

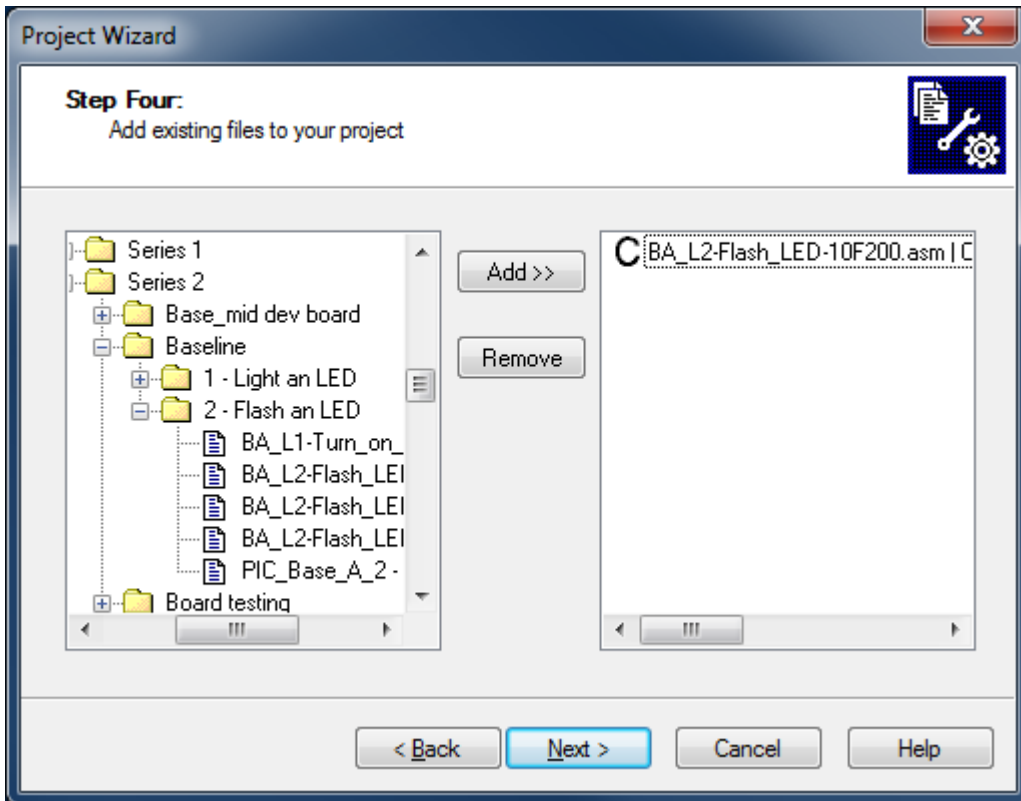
The correct device (PIC10F200 or PIC12F508) will already be selected, as will the toolsuite (MPASM), so simply click “Next” until you get to Step Three, and select “Reconfigure Active Project” and “Make changes without saving” and “Make changes without saving”, as shown:



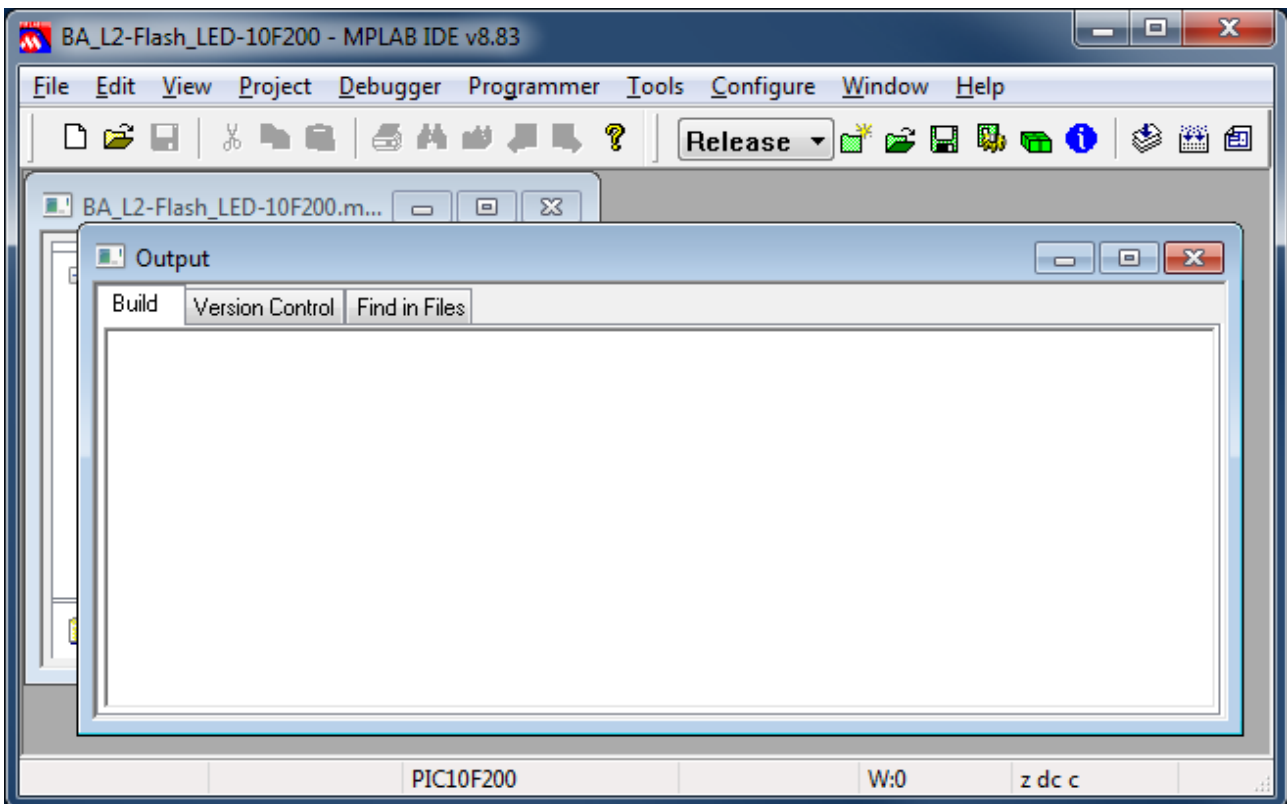
You are now presented with the following window, showing the assembler source file with a “U” to indicate a user file, in the new project directory, but with the same name as before:



Click on the “U” until it changes to a “C”. You can now click on the file name and rename it to something more appropriate to this lesson, such as ‘BA_L2-Flash_LED.asm’:

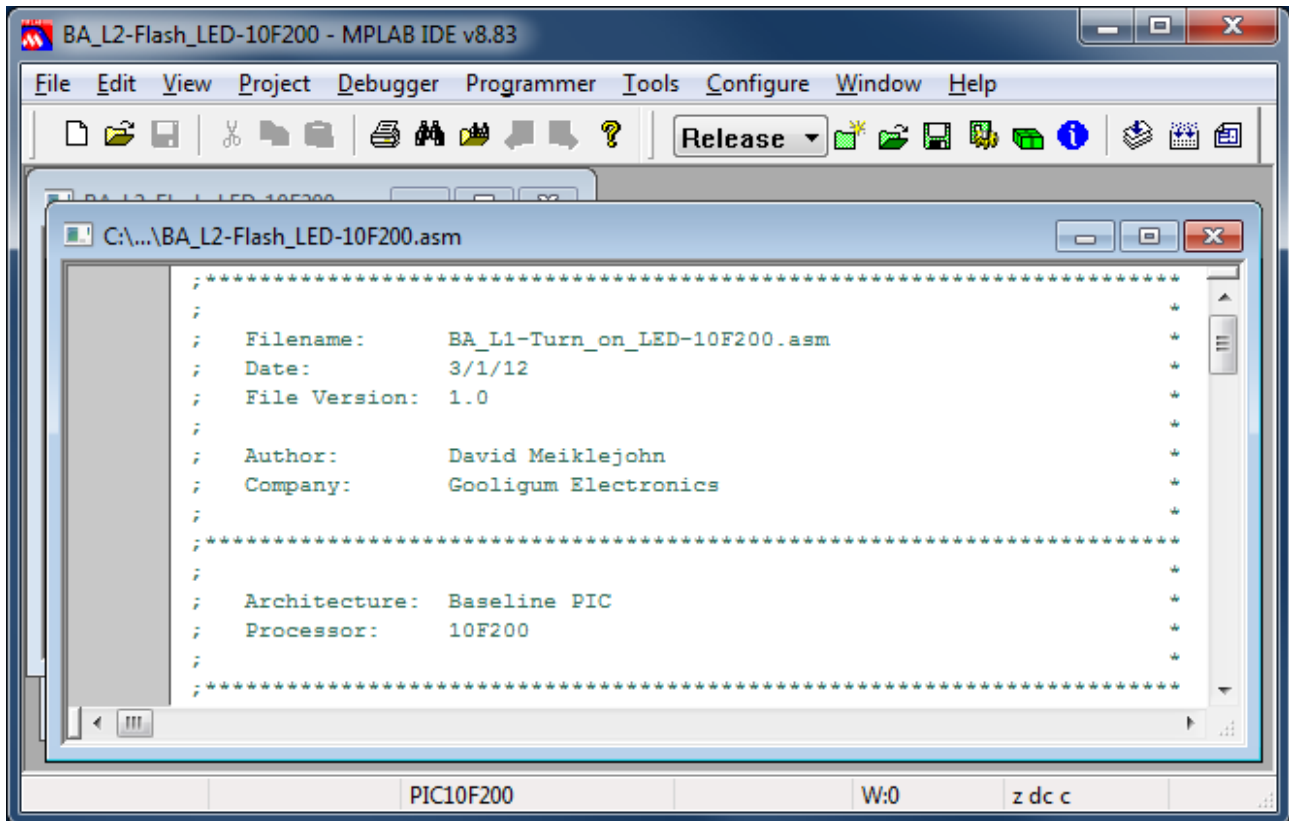


Finally click “Next” then “Finish” and the project is reconfigured, with the renamed source file:



It's a good idea at this point to save your new project, using the "Save Workspace" icon, or the "Project → Save Project" menu item.

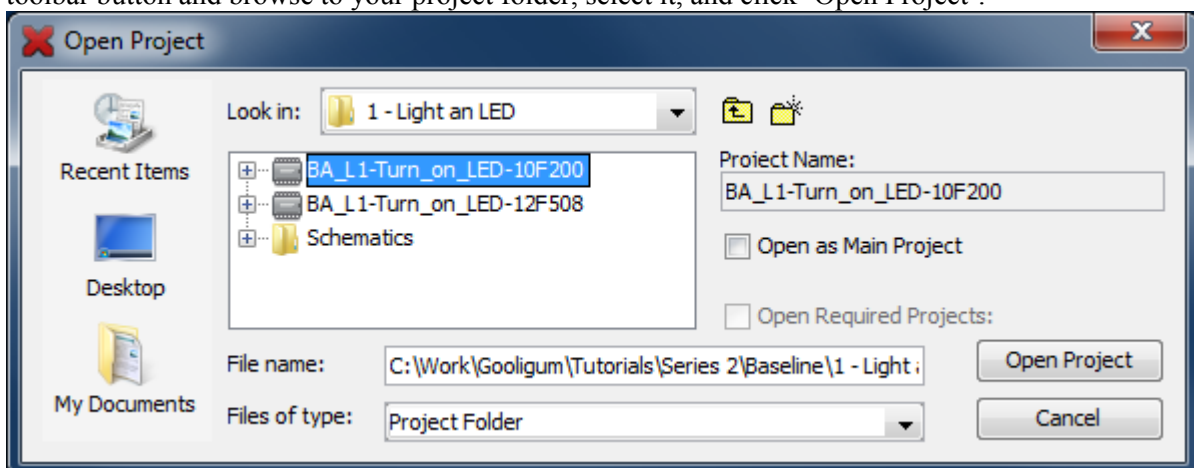
If you double-click on the source file ('BA_L2-Flash_LED-10F200.asm' in this example), you'll see a copy of your code from lesson 1:



MPLAB X

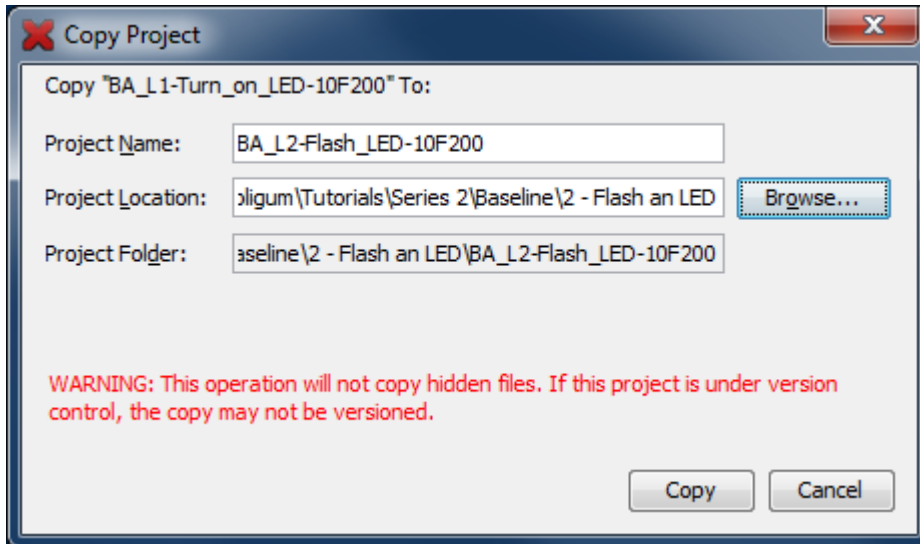
To create a new project in MPLAB X, based on an existing project, you first need to run MPLAB X (you can't simply double-click on a project file, like you can with MPLAB 8 projects), and then open your existing project within MPLAB X.

If you were recently working on the project you want to copy (such as the project from lesson 1), it is probably already visible in the Projects window. If it's not, it may appear under the "File → Open Recent Project" menu list. Or you can use the "File → Open Project" menu item, or click on the "Open Project..." toolbar button and browse to your project folder, select it, and click 'Open Project':



You should now right-click the project name ('BA_L1-Turn_on_LED-10F200' in this example) in the Projects window, and select "Copy...".

The "Copy Project" dialog now gives you a chance to give your copied project a new name, such as 'BA_L2-Flash_LED'. You can also specify (and create, if you wish) a new folder for this project, by browsing to it:

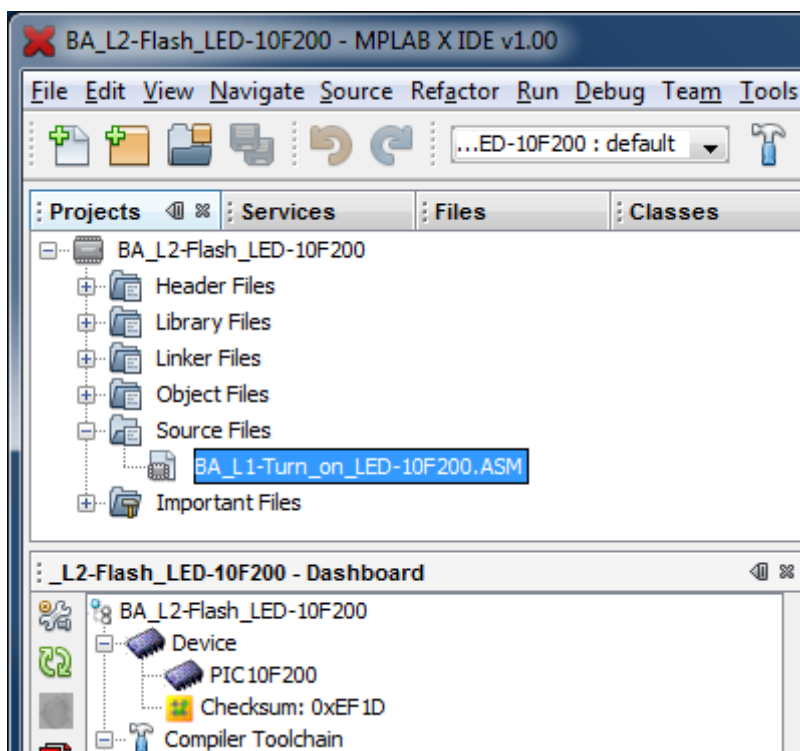


When you are satisfied with your new project name and location, click 'Copy'.

Your new project should now appear in the Projects window.

You can close your old project by right-clicking it and selecting "Close", so that only your new project is visible.

If you expand your new project, you'll see that source file from the old project has been copied into the new project, with its original name:



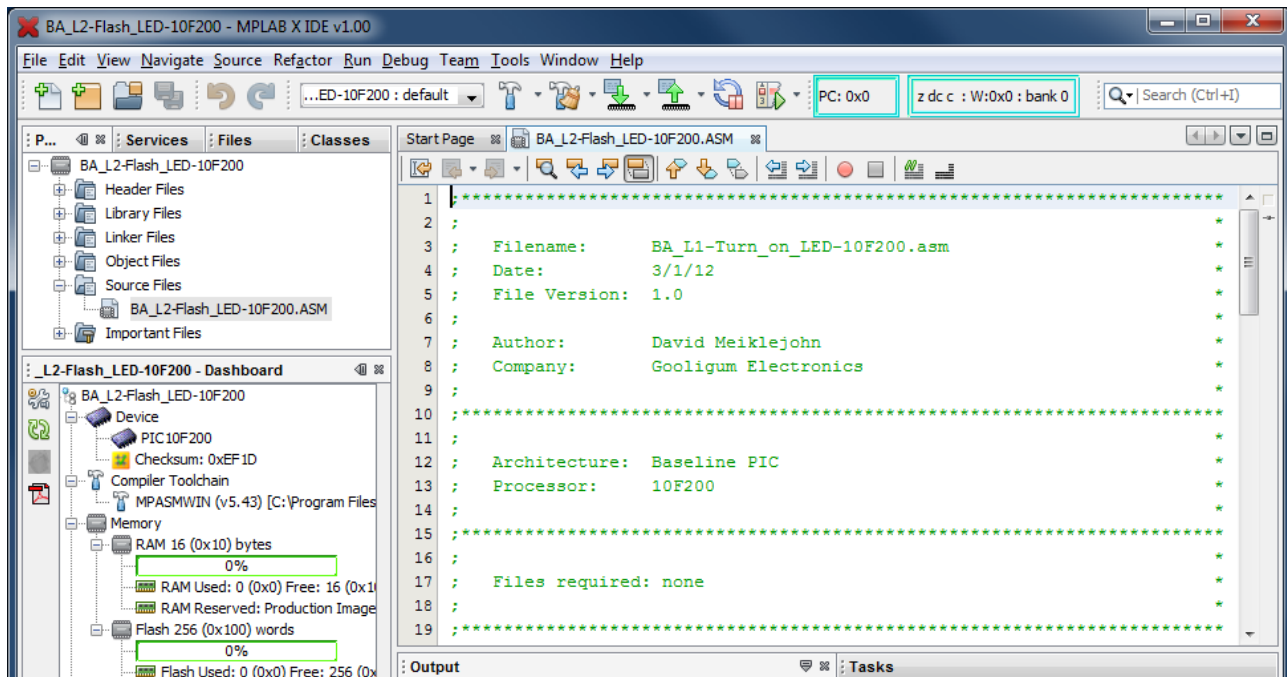
To rename the source file, to something more appropriate for this project, right-click it and select "Rename...".

Type in the new name, such as 'BA_L2-Flash_LED' and then click 'OK'.

Note that there is no need to type the '.ASM' suffix – the Rename dialog will keep the existing file extension.

You now have a new project, with a new name in a new location, with a renamed source file, copied from your old project.

If you double-click your new source file, you'll see a copy of your code from lesson 1 in an editor window:



Flashing the LED

Whether you are using MPLAB 8 or X, you can now use the editor to update your code from lesson 1.

We'll need to add some code to make the LED flash, but first the comments should be updated to reflect the new project. For example:

```
;*****
;
;  Filename:      BA_L2-Flash_LED-10F200.asm
;  Date:         20/1/12
;  File Version: 1.0
;
;  Author:       David Meiklejohn
;  Company:      Gooligum Electronics
;
;*****
;
;  Architecture: Baseline PIC
;  Processor:    10F200
;
;*****
;
;  Files required: none
;
;*****
;
;  Description:   Lesson 2, example 1
;
;  Flashes a LED at approx 1 Hz.
;  LED continues to flash until power is removed.
;
;*****
;
```

```

; Pin assignments:
; GP1 = flashing LED
;
;*****

```

We're using the same PIC device as before, and it will be configured the same way, so we can leave the processor definition and configuration sections unchanged. There is also no need to change the internal RC oscillator calibration or reset vector sections.

So, for the PIC10F200 version, we still have, unchanged from lesson 1:

```

list      p=10F200
#include   <p10F200.inc>

;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog
__CONFIG     _MCLRE_ON & _CP_OFF & _WDT_OFF

;***** RC CALIBRATION
RCCAL CODE    0x0FF          ; processor reset vector
            res 1           ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET CODE    0x000          ; effective reset vector
            movwf OSCCAL     ; apply internal RC factory calibration

```

while for the PIC12F508, we have instead (also unchanged from lesson 1):

```

list      p=12F508
#include   <p12F508.inc>

;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, int RC clock
__CONFIG     _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC

;***** RC CALIBRATION
RCCAL CODE    0x1FF          ; processor reset vector
            res 1           ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET CODE    0x000          ; effective reset vector
            movwf OSCCAL     ; apply internal RC factory calibration

```

Again, we need to set up the PIC so that only GP1 is configured as an output, so we can leave the initialisation code from lesson 1 intact:

```

;***** MAIN PROGRAM *****

;***** Initialisation
start
            movlw  b'111101'    ; configure GP1 (only) as an output
            tris   GPIO

```

In [lesson 1](#), we made GP1 high, and left it that way. To make it flash, we need to set it high, then low, and then repeat. You may think that you could achieve this with something like:

```
flash
    movlw    b'000010'        ; set GP1 high
    movwf    GPIO
    movlw    b'000000'        ; set GP1 low
    movwf    GPIO
    goto     flash            ; repeat forever
```

If you try this code, you'll find that the LED appears to remain on continuously. In fact, it's flashing too fast for the eye to see.

Our PIC is using an internal RC oscillator¹, clocked at a nominal 4 MHz. Each instruction executes in four clock cycles, or 1 μ s – except instructions which branch to another location, such as 'goto', which require two instruction cycles, or 2 μ s².

This loop takes a total of 6 μ s, so the LED flashes at $1/(6 \mu\text{s}) = 166.7$ kHz. That's much too fast to see!

To slow it down to a more sedate (and visible!) 1 Hz, we have to add a delay. But before looking at delays, we can make a small improvement to the code.

To flip, or toggle, a single bit – to change it from 0 to 1 or from 1 to 0, you can exclusive-or it with 1.

That is:

$$0 \text{ XOR } 1 = 1$$

$$1 \text{ XOR } 1 = 0$$

So to repeatedly toggle GP1, we can read the current state of GPIO, exclusive-or the bit corresponding to GP1, then write it back to GPIO, as follows:

```
flash
    movlw    b'000010'        ; bit mask to toggle GP1 only
    xorwf    GPIO,f           ; toggle GP1 using mask in W
    goto     flash            ; repeat forever
```

The 'xorwf' instruction exclusive-ors the W register with the specified register – “exclusive-or W with file register”, and writes the result either to the specified file register (GPIO in this case) or to W.

Note that there is no need to set GP1 to an initial state; whether it's high or low to start with, it will be successively flipped.

Many of the PIC instructions, like xorwf, are able to place the result of an operation (e.g. add, subtract, or in this case XOR) into either a file register or W. This is referred to as the instruction destination. A ', f' at the end indicates that the result should be written back to the file register; to place the result in W, use ', w' instead.

This single instruction – 'xorwf GPIO, f' – is doing a lot of work. It reads GPIO, performs the XOR operation, and then writes the result back to GPIO.

The read-modify-write problem

And therein lays a potential problem. You'll find it referred to as the *read-modify-write* problem. When an instruction reads a port register, such as GPIO, the value that is read back is not necessarily the value that

¹ The 12F508 has been configured (using the `__config` directive) to use its internal RC oscillator, while the 10F200 can only use an internal RC oscillator; there is no other choice.

² Assuming a 4 MHz processor clock

you originally wrote to it. When the PIC reads a port register, it doesn't read the value in the "output latch" (i.e. the value you wrote to it). Instead, it reads the pins themselves – the voltages present in the circuit.

Normally, that doesn't matter. When you write a '1', the corresponding pin (if configured as an output) will go to a high voltage level, and when you then read that pin, it's still at a high voltage, so it reads back as a '1'. But if there's excessive load on that pin, the PIC may not be able to drive it high, and it will read as a '0'. Or capacitance loading the output line may mean a delay between the PIC's attempt to raise the voltage and the voltage actually swinging high enough to register as a '1'. Or noise in the circuit may mean that a line that normally reads as a '1', sometimes (randomly) reads as a '0'.

In this simple case, particularly when we slow the flashing down to 1 Hz, you'll find that this isn't an issue. The above code will usually work correctly. But it's good to get into good habits early. For the reasons given above, it is considered "bad practice" to assume a value you have previously written is still present on an I/O port register.

It's better to keep a copy of what the port value is supposed to be, and operate on that, then copy it to the port register. This is referred to as using a *shadow register*.

We could use *W* as a shadow register, as follows:

```

        movlw    b'000000'      ; start with W zeroed
flash
        xorlw    b'000010'      ; toggle W bit corresponding to GP1 (bit 1)
        movwf    GPIO           ; and write to GPIO
        goto     flash          ; repeat forever

```

Each time around the loop, the contents of *W* are updated and then written to the I/O port.

The '`xorlw`' instruction exclusive-ors a literal value with the *W* register, placing the result in *W* – "exclusive-or literal to **W**".

Normally, instead of '`movlw b'000000'`' (or simply '`movlw 0`') you'd use the '`clrw`' instruction – "clear **W**".

'`clrw`' has the same effect as '`movlw 0`', except that '`clrw`' sets the '**Z**' (zero) status flag, while the '`movlw`' instruction doesn't affect any of the status flags, including **Z**.

Status flags are bits in the **STATUS** register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	GPWUF	-	-	\overline{TO}	\overline{PD}	Z	DC	C

Certain arithmetic or logical operations will set or clear the **Z**, **DC** or **C** status bits, and other instructions can test these bits, and take different actions depending on their value. We'll see examples of testing these flags in later lessons.

We're not using **Z** here, so we can use `clrw` to make the code more readable:

```

        clr     W               ; use W to shadow GPIO - initially zeroed
flash
        xorlw    b'000010'      ; toggle W bit corresponding to GP1 (bit 1)
        movwf    GPIO           ; and write to GPIO
        goto     flash          ; repeat forever

```

It would be very unusual to be able to use *W* as a shadow register, because it is used in so many PIC instructions. When we add delay code, it will certainly need to be able to change the contents of *W*, so we'll have to use a file register to hold the shadow copy of **GPIO**.

In [lesson 1](#), we saw how to allocate data memory for variables (such as shadow registers), using the `UDATA` and `RES` directives. In this case, we need something like:

```
;***** VARIABLE DEFINITIONS
      UDATA
sGPIO  res 1          ; shadow copy of GPIO
```

The flashing code now becomes:

```
      clrfsf  sGPIO          ; clear shadow register
flash
      movf    sGPIO,w        ; get shadow copy of GPIO
      xorlw   b'000010'     ; toggle bit corresponding to GP1 (bit 1)
      movwf   sGPIO         ; in shadow register
      movwf   GPIO          ; and write to GPIO
      goto   flash         ; repeat forever
```

That's nearly twice as much code as the first version, that operated on `GPIO` directly, but this version is much more robust.

There are two new instructions here.

'`clrfsf`' clears (sets to 0) the specified register – “**clear file register**”.

'`movf`', with '`,w`' as the destination, copies the contents of the specified register to `W` – “**move file register to destination**”. This is the instruction used to read a register.

'`movf`', with '`,f`' as the destination, copies the contents of the specified register to itself. That would seem to be pointless; why copy a register back to itself? The answer is that the '`movf`' instruction affects the `Z` status flag, so copying a register to itself is a sneaky way to test whether the value in the register is zero.

Delay Loops

To make the flashing visible, we need to slow it down, and that means getting the PIC to “do nothing” between LED changes.

The baseline PICs do have a “do nothing” instruction: '`nop`' – “**no operation**”. All it does is to take some time to execute.

How much time depends on the clock rate. Instructions are executed at one quarter the rate of the processor clock. In this case, the PIC is using the internal RC clock, running at a nominal 4 MHz (see [lesson 1](#)). The instructions are clocked at ¼ of this rate: 1 MHz. Each instruction cycle is then 1 µs.

Most baseline PIC instructions, including '`nop`', execute in a single cycle. The exceptions are those which jump to another location (such as '`goto`') or if an instruction is conditionally skipped (we'll see an example of this soon). So '`nop`' provides a 1 µs delay – not very long!

Another “do nothing” instruction is '`goto $+1`'. Since '\$' stands for the current address, '\$+1' is the address of the next instruction. Hence, '`goto $+1`' jumps to the following instruction – apparently useless behaviour. But all '`goto`' instructions executes in two cycles. So '`goto $+1`' provides a 2 µs delay in a single instruction – equivalent to two '`nop`'s, but using less program memory.

To flash at 1 Hz, the PIC should light the LED, wait for 0.5 s, turn off the LED, wait for another 0.5 s, and then repeat.

Our code changes the state of the LED once each time around the loop, so we need to add a delay of 0.5 s within the loop. That's 500,000 μ s, or 500,000 instruction cycles. Clearly we can't do that with 'nop's or 'goto's alone!

The answer, of course, is to use loops to execute instructions enough times to build up a useful delay. But we can't just use a 'goto', or else it would loop forever and the delay would never finish. So we have to loop some finite number of times, and for that we need to be able to count the number of times through the loop (incrementing or decrementing a loop counter variable) and test when the loop is complete.

Here's an example of a simple "do nothing" delay loop:

```

        movlw    .10
        movwf   dc1           ; dc1 = 10 = number of loop iterations
dly1   nop
        decfsz  dc1, f
        goto    dly1

```

The first two instructions write the decimal value "10" to a loop counter variable called 'dc1'.

Note that to specify a decimal value in MPASM, you prefix it with a '.'. If you don't include the '.', the assembler will use the default radix (hexadecimal), and you won't be using the number you think you are! Although it's possible to set the default radix to decimal, you'll run into problems if you rely on a particular default radix and then later copy and paste your code into another project, with a different default radix, giving different results. It's much safer, and clearer, to simply prefix all hexadecimal numbers with '0x' and all decimal numbers with '.'.

The 'decfsz' instruction performs the work of implementing the loop – “**d**ecre**ment** **f**ile register, **s**kip if **z**ero”. First, it decrements the contents of the specified register, writes the result back to the register (as specified by the 'f' destination), then tests whether the result was zero. If it's not yet zero, the next instruction is executed, which will normally be a 'goto' which jumps back to the start of the loop. But if the result of the decrement is zero, the next instruction is skipped; since this is typically a 'goto', skipping it means exiting the loop.

The 'decfsz' instruction normally executes in a single cycle. But if the result is zero, and the next instruction is skipped, an extra cycle is added, making it a two-cycle instruction.

There is also an 'incfsz' instruction, which is equivalent to 'decfsz', except that it increments instead of decrementing. It's used if you want to count up instead of down. For a loop with a fixed number of iterations, counting down is more intuitive than counting up, so 'decfsz' is more commonly used for this.

In the code above, the loop counter, 'dc1', starts at 10. At the end of the first loop, it is decremented to 9, which is non-zero, so the 'goto' instruction is not skipped, and the loop repeats from the 'dly1' label. This process continues – 8,7,6,5,4,3,2 and on the 10th iteration through the loop, dc1 = 1. This time, dc1 is decremented to zero, and the “skip if zero” comes into play. The 'goto' is skipped, and execution continues after the loop.

You can see that the number of loop iterations is equal to the initial value of the loop counter (10 in this example). Call that initial number N. The loop executes N times.

To calculate the total time taken by the loop, add the execution time of each instruction in the loop:

nop		1
decfsz	dc1, f	1 (except when result is zero)
goto	dly1	2

That's a total of 4 cycles, except the last time through the loop, when the `decfsz` takes an extra cycle and the `goto` is not executed (saving 2 cycles), meaning the last loop iteration is 1 cycle shorter. And there are two instructions before the loop starts, adding 2 cycles.

Therefore the total delay time = $(N \times 4 - 1 + 2)$ cycles = $(N \times 4 + 1)$ μ s

If there was no 'nop', the delay would be $(N \times 3 + 1)$ μ s; if two 'nop's, then it would be $(N \times 5 + 1)$ μ s, etc.

It may seem that, because 255 is the highest 8-bit number, the maximum number of iterations (N) should be 255. But not quite. If the loop counter is initially 0, then the first time through the loop, the 'decfsz' instruction will decrement it, and if an 8-bit counter is decremented from 0, the result is 255, which is non-zero, and the loop continues – another 255 times. Therefore the maximum number of iterations is in fact 256, with the loop counter initially 0.

So for the longest possible single loop delay, we can write something like:

```

                clrfsz   dc1           ; loop 256 times
dly1           nop
                decfsz   dc1, f
                goto     dly1

```

The two "move" instructions have been replaced with a single 'clrfsz', using 1 cycle less, so the total time taken is $256 \times 4 = 1024$ μ s \approx 1 ms.

That's still well short of the 0.5 s needed, so we need to wrap (or *nest*) this loop inside another, using separate counters for the inner and outer loops, as shown:

```

                movlw    .200           ; loop (outer) 200 times
                movwf    dc2
                clrfsz   dc1           ; loop (inner) 256 times
dly1           nop                    ; inner loop = 256 x 4 - 1 = 1023 cycles
                decfsz   dc1, f
                goto     dly1
                decfsz   dc2, f
                goto     dly1

```

The loop counter 'dc2' is being used to control how many times the inner loop is executed.

Note that there is no need to clear the inner loop counter (dc1) on each iteration of the outer loop, because every time the inner loop completes, `dc1 = 0`.

The total time taken for each iteration of the outer loop is 1023 cycles for the inner loop, plus 1 cycle for the 'decfsz dc2, f' and 2 cycles for the 'goto' at the end, except for the final iteration, which, as we've seen, takes 1 cycle less. The three setup instructions at the start add 3 cycles, so if the number of outer loop iterations is N:

Total delay time = $(N \times (1023 + 3) - 1 + 3)$ cycles = $(N \times 1026 + 2)$ μ s.

The maximum delay would be with $N = 256$, giving 262,658 μ s. We need a bit less than double that. We could duplicate all the delay code, but it takes fewer lines of code if we duplicate only the inner loop:

```

                ; delay 500ms
                movlw    .244           ; outer loop: 244 x (1023 + 1023 + 3) + 2
                movwf    dc2           ; = 499,958 cycles
                clrfsz   dc1           ; inner loop: 256 x 4 - 1
dly1           nop                    ; inner loop 1 = 1023 cycles
                decfsz   dc1, f
                goto     dly1
dly2           nop                    ; inner loop 2 = 1023 cycles
                decfsz   dc1, f
                goto     dly2
                decfsz   dc2, f
                goto     dly1

```

The two inner loops of 1023 cycles each, plus the 3 cycles for the outer loop control instructions (`decfsz` and `goto`) make a total of 2049 μ s. Dividing this into 500,000 gives 244.02 – pretty close to a whole number, so an outer loop count of 244 will be very close to what’s needed.

The calculations are shown in the comments above. The total time for this delay code is 499,958 cycles. In theory, that’s 499.958 ms – within 0.01% of the desired result! Given that that’s much more accurate than the 4 MHz internal RC oscillator, there is no point trying for more accuracy than this.

But suppose the calculation above had come out as needing some fractional number of outer loop iterations, say 243.5 – what would you do? Generally you’d fine-tune the timing by adding or removing ‘`nop`’s. E.g. suppose that both inner loops had 2 ‘`nop`’s instead of 1. Then they would execute in $256 \times 5 - 1 = 1279$ cycles, and the calculation for the outer loop counter would be $500,000 \div (1279 + 1279 + 3) = 195.24$. That’s not as good a result as the one above, because ideally we want a whole number of loops. 244.02 is much closer to being a whole number than 195.24.

For even finer control, you can add ‘`nop`’s to the outer loop, immediately before the ‘`decfsz dc2, f`’ instruction. One extra ‘`nop`’ would give the outer loop a total of $1023 + 1023 + 4 = 2050$ cycles, instead of 2049. The loop counter calculation becomes $500,000 \div 2050 = 243.90$. That’s not bad, but 244.02 is better, so we’ll leave the code above unchanged.

With a bit of fiddling, once you get some nested loops close to the delay you need, adding or removing ‘`nop`’ or ‘`goto $+1`’ instructions can generally get you quite close to the delay you need. And remember that it is pointless to aim for high precision (< 1%) when using the internal RC oscillator. When using a crystal, it makes more sense to count every last cycle accurately, as we’ll see in [lesson 7](#).

For delays longer than about 0.5 s, you’ll need to add more levels of nesting to your delay loops – with enough levels you can count for years!

Complete program

Putting together all these pieces, here’s the complete PIC10F200 version of our LED flashing program:

```

;*****
;
; Description: Lesson 2, example 1
;
; Flashes a LED at approx 1 Hz.
; LED continues to flash until power is removed.
;
;*****
;
; Pin assignments:
; GP1 = flashing LED
;
;*****

list      p=10F200
#include  <p10F200.inc>

;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog
__CONFIG      _MCLRE_ON & _CP_OFF & _WDT_OFF

;***** VARIABLE DEFINITIONS
                UDATA
sGPIO         res 1                ; shadow copy of GPIO

```

```

dc1      res 1          ; delay loop counters
dc2      res 1

;***** RC CALIBRATION
RCCAL    CODE    0x0FF      ; processor reset vector
          res 1          ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET    CODE    0x000      ; effective reset vector
          movwf   OSCCAL     ; apply internal RC factory calibration

;***** MAIN PROGRAM *****

;***** Initialisation
start
          movlw   b'111101'   ; configure GP1 (only) as an output
          tris    GPIO

          clrf    sGPIO       ; start with shadow GPIO zeroed

;***** Main loop
main_loop
          ; toggle LED on GP1
          movf    sGPIO,w      ; get shadow copy of GPIO
          xorlw   b'000010'    ; toggle bit corresponding to GP1 (bit 1)
          movwf   sGPIO       ; in shadow register
          movwf   GPIO        ; and write to GPIO

          ; delay 500ms
          movlw   .244         ; outer loop: 244 x (1023 + 1023 + 3) + 2
          movwf   dc2         ; = 499,958 cycles
          clrf    dc1         ; inner loop: 256 x 4 - 1
dly1     nop                 ; inner loop 1 = 1023 cycles
          decfsz  dc1,f
          goto    dly1
dly2     nop                 ; inner loop 2 = 1023 cycles
          decfsz  dc1,f
          goto    dly2
          decfsz  dc2,f
          goto    dly1

          goto    main_loop    ; repeat forever

          END

```

The 12F508 version is very similar, with changes to the `list`, `#include`, `__CONFIG` and `RCCAL CODE` directives, as shown earlier.

If you follow the programming procedure described in [lesson 1](#), you should now see your LED flashing at something very close to 1 Hz.

Conclusion

It's taken two lessons and dozens of pages to get here, but we finally have a flashing LED!

In this lesson, we built on the first, showing how to base a new project on an existing one, modifying it and adding whatever additional features the new project needs.

We saw how to toggle a pin, discussed how “read-modify-write” operations on a port can be problematic, and showed how to use shadow registers can be used to avoid such potential problems.

We also saw how to use decrement instructions with conditional tests to implement loops, and how to use loops to create delays of any length.

In the [next lesson](#) we'll step up to a slightly bigger PIC, the 12F509.

We'll also see how to make our programs more modular, so that useful pieces of code such as the 500 ms delay developed here can be easily re-used.

Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 3: Introducing Modular Code

[Lesson 2](#) introduced delay loops, which we used in flashing an LED.

Delay loops are an example of useful pieces of code that could be re-used in other applications; you don't want to have to re-invent the wheel (or delay loop!) every time. Or, in a larger application, you may need to use delays in several parts of the program. It would be wasteful to have to include multiple copies of the same code in the one program. And if you wanted to make a change to your delay code, it would be not only more convenient, but less likely to introduce errors, if you only have to change it in one place.

Code that is made up of pieces that can be easily re-used, either within the same program, or in other programs, is called modular. You'll save yourself a lot of time if you learn to write re-usable, modular code, which is why it's being covered in such an early lesson, even though these techniques are most useful in larger programs.

As your programs become larger and more complex, you'll need a PIC with more memory than the 10F200 or 12F508 we've seen so far. Unfortunately the baseline PIC architecture has some limitations which need to be taken in to account when working with devices with more memory, and it's very important to learn how techniques such as banking and paging are used properly access data and program memory in larger PICs. We'll need a bigger baseline PIC to learn these techniques with, so this lesson will also introduce the PIC12F509.

In this lesson, we will learn about:

- The PIC12F509 MCU
- Subroutines
- Banking and paging
- Relocatable code
- External modules

We'll continue to assume that you're using either the [Gooligum Baseline and Mid-range PIC Training and Development Board](#) or Microchip's Low Pin Count (LPC) Demo Board, with Microchip's MPLAB 8 or MPLAB X integrated development environment and a Microchip PICkit 2 or PICkit 3 programmer – see [lesson 1](#) for details.

Introducing the PIC12F509

As we saw in lesson 1, the 12F509 is essentially a 12F508 with more memory.

It comes in the same packages, with the same pin-out and number of I/O pins, has the same peripherals (such as timers; see [lesson 5](#)) and runs at the same clock speed.

However, the 12F509 has twice the program memory (1024 words instead of 512) and more data memory (41 bytes instead of 25):

Device	Program Memory (words)	Data Memory (bytes)	Package	I/O pins	Clock rate (maximum)
12F508	512	25	8-pin	6	4 MHz
12F509	1024	41	8-pin	6	4 MHz
16F505	1024	72	14-pin	12	20 MHz

Banking

There's a problem with having extra data memory: baseline PIC instructions can only directly access, or *address*, a small number of registers.

At the lowest level, PIC instructions consist of bits. In the baseline PIC architecture, each instruction word is twelve bits wide. Some of these bits designate which instruction it is; this set of bits is called the *opcode*.

For example, the opcode for `movlw` is 1100.

The remaining bits in each 12-bit instruction word are used to specify whatever value is associated with that instruction, such as a literal value, a register, or an address. In the case of `movlw`, the opcode is four bits long, leaving the other eight bits to hold the literal value that will be moved into *W*.

Thus, the 12-bit instruction word for '`movlw 1`' is 1100 00000001 in binary, the first four bits meaning '`movlw`' and the last eight bits being the binary for '1'.

In the baseline architecture, only five bits are allocated to register addressing.

For example, the opcode for `clrf` is 0000011, which is seven bits long, and the remaining five bits specify which register is to be acted on (cleared, in this case).

The program code in the [last lesson](#) included the instruction '`clrf dc1`', where `dc1` is a variable, stored in one of the PIC's general purpose registers.

Although it's easiest for us to use names (such as '`dc1`') for the variables in our programs, the PIC really only knows about numbers: each register having its own number, or *address*.

When our project is built, the linker assigns an address to every variable. All the names, such as variables and program labels, in our source code are replaced with numeric addresses assigned by the linker, before the assembled code is loaded into the PIC and run.

Suppose that the linker decides that the variable '`dc1`' should be stored in the register at address 20. After being assembled and linked, our source code of '`clrf dc1`' would end up as the 12-bit binary instruction 0000011 10100, where the first seven bits mean '`clrf`' and the remaining five bits are '20' in binary.

Five bits is enough to allow up to 32 registers to be directly addressed, numbered from 0 to 31.

This is called a *register bank*.

We saw in [lesson 1](#) that the 12F508 has a total of 32 registers (exactly one full bank), consisting of 7 special function registers, such as `STATUS` and `GPIO`, followed by 25 general purpose registers, which can be used to store variables.

That's exactly one bank of registers, as much as any baseline instruction can directly access.

PIC12F509 Registers

Address	Bank 0	Address	Bank 1
00h	INDF	20h	INDF
01h	TMR0	21h	TMR0
02h	PCL	22h	PCL
03h	STATUS	23h	STATUS
04h	FSR	24h	FSR
05h	OSCCAL	25h	OSCCAL
06h	GPIO	26h	GPIO
07h	General Purpose Registers	27h	Map to Bank 0 07h – 0Fh
0Fh			
10h		General Purpose Registers	
1Fh			
	30h		General Purpose Registers
	3Fh		

The 12F509 has 41 general purpose registers, in addition to the 7 special function registers; 48 in total. That’s too many to fit into a single bank.

To allow these additional registers to be addressed, they are arranged into two banks, as shown on the right.

The bank to be accessed is selected by bit 5 in the FSR register (FSR<5>). If it is cleared to ‘0’, bank 0 is selected, and any instructions which reference a register will address a register in bank 0. If FSR<5> is set to ‘1’, bank 1 is selected, and subsequent instructions will reference registers in bank 1.

The special function registers appear in both banks. Regardless of which bank is selected, you can refer directly to any special function register, such as GPIO¹.

The first set of nine general purpose registers (07h – 0Fh) are mapped into both banks. Whichever bank is selected, these same registers will be addressed. Registers like this, which appear at the same location across all banks, are referred to as *shared*. They are very useful for storing data or variables which you want to access often, regardless of which bank is selected, without having to include bank selection instructions. If you address a register as 07h or 27h, it will contain the same data; it’s the same physical register.

The next 16 general purpose registers (10h – 1Fh) are accessed through bank 0 only. If you set FSR<5> to select bank 1, you’ll access an entirely separate set of 16 general purpose registers (30h – 3Fh)².

Thus, the 12F509 has 9 shared general purpose registers, and 32 banked general purpose registers (16 in each of two banks), for a total of 41 bytes of data memory.

Taking this banking scheme further, the 16F505 has 72 bytes of data memory, arranged into four banks: 8 shared registers and 64 banked registers (16 in each bank). As in the other baseline devices, the special function registers are mapped into each bank.

¹ That’s not true in the midrange devices, where you have to be very careful to select the correct bank before accessing special function registers.

² When referring to numeric register addresses, FSR<5> is considered to be bit 5 of the register address, with bits 0 to 4 of the address coming from the instruction word.

The four data banks in the 16F505 are selected by bits 5 and 6 of the FSR register (FSR<6:5>): ‘00’ selects bank 0, ‘01’ for bank 1, ‘10’ for bank 2, and ‘11’ selects bank 3.

Similarly, the 16F59 has 134 general purpose registers: 6 shared and 16 in each of 8 banks. To specify which of the eight banks is selected, three bits are needed: FSR<7:5>.

Since the FSR has only eight bits, this scheme can’t be extended any further, so eight is the maximum number of data banks possible in the midrange architecture.

Later in this lesson, under “Using the BANKSEL directive”, we’ll see how, and when, to correctly specify these bank selection bits.

Paging

A similar problem exists with addressing program memory.

As discussed above, baseline PIC instructions are twelve bits wide and consist of an opcode, designating the instruction, with the remaining bits specifying the a value, such as a register address.

The opcode for `goto` is 101. That’s three bits, leaving nine bits to specify the address to jump to.

Nine bits are enough to specify any value from 0 to 511. That’s 512 addresses in all.

This is called a *page* of program memory.

The program memory on the 12F508 is 512 words, which is exactly one page. Since the `goto` instruction can specify any of these 512 addresses, it can be used to jump anywhere in the 12F508’s memory, directly.

That’s fine for the 12F508, but it’s a problem for a device such as the 12F509, with 1024 words.

The solution is to use a bit in the STATUS register, PA0, to select which page is to be accessed:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	GPWUF	-	PA0	\overline{TO}	\overline{PD}	Z	DC	C

The *program counter* (PC) holds the full 12-bit address of the next instruction to be executed. Whenever a `goto` instruction is executed, the lower 9 bits of the program counter (PC<8:0>) are taken from the `goto` instruction word, but the 10th bit (PC<9>) is provided by the current value to PA0.

Therefore, to `goto` an address in the first 512 words of program memory (page 0), you must first clear PA0. To jump to code in page 1, you must first set PA0 to ‘1’.

If you don’t update PA0, but then try to `goto` an address in a different page, you will instead jump to the corresponding address in the current page – not the location you were trying to access, and your program will almost certainly fail.

For baseline devices with 2048 words of program memory, such as the 16F59, this paging scheme is extended, with bit 6 of the STATUS register, referred to as PA1, providing PC<10>. Given two page selection bits (PA0 and PA1), up to four 512-word pages can be selected, allowing a total of 2048 words.

We’ll see in the “Using the PAGESEL directive” section, later in this lesson, when and how to correctly specify these page selection bits.

Subroutines

Here again is the main program code from [lesson 2](#):

```

;***** Initialisation
start
    movlw    b'111101'        ; configure GP1 (only) as an output
    tris     GPIO

    clrf     sGPIO           ; start with shadow GPIO zeroed

;***** Main loop
main_loop
    ; toggle LED on GP1
    movf     sGPIO,w         ; get shadow copy of GPIO
    xorlw    b'000010'       ; toggle bit corresponding to GP1 (bit 1)
    movwf    sGPIO          ; in shadow register
    movwf    GPIO           ; and write to GPIO

    ; delay 500ms
    movlw    .244            ; outer loop: 244 x (1023 + 1023 + 3) + 2
    movwf    dc2             ; = 499,958 cycles
    clrf     dc1             ; inner loop: 256 x 4 - 1
dly1        nop              ; inner loop 1 = 1023 cycles
            decfsz    dc1,f
            goto     dly1
dly2        nop              ; inner loop 2 = 1023 cycles
            decfsz    dc1,f
            goto     dly2
            decfsz    dc2,f
            goto     dly1

    goto     main_loop       ; repeat forever

END

```

Suppose that you wanted to include another 500 ms delay in another part of the program. To place the delay code *inline*, as it is above, would mean repeating all 11 lines of the delay routine somewhere else. And you have to be very careful when copying and pasting code – you can't refer to the labels 'dly1' or 'dly2' in the copied code, or else it will jump back to the original delay routine – probably not the intended effect!

The usual way to use the same code in a number of places in a program is to place it into a *subroutine*. The main code loop would then something look like this:

```

main_loop
    movf     sGPIO,w         ; get shadow copy of GPIO
    xorlw    b'000010'       ; toggle bit corresponding to GP1 (bit 1)
    movwf    sGPIO          ; in shadow register
    movwf    GPIO           ; and write to GPIO
    call     delay500        ; delay 500ms
    goto     main_loop       ; repeat forever

```

The 'call' instruction – “**call** subroutine” – is similar to 'goto', in that it jumps to another program address. But first, it copies (or *pushes*) the address of the next instruction onto the stack.

The *stack* is a set of registers, used to hold the return addresses of subroutines. When a subroutine is finished, the *return address* is copied (*popped*) from the stack to the program counter, and program execution continues with the instruction following the subroutine call.

The baseline PICs only have two stack registers, so a maximum of two return addresses can be stored. This means that you can call a subroutine from within another subroutine, but you can't *nest* the subroutine calls

any deeper than that. But for the sort of programs you'll want to write on a baseline PIC, you'll find this isn't usually a problem. If it is, then it's time to move up to a mid-range PIC, or a PIC18...

The instruction to *return* from a subroutine is 'retlw' – "**return with literal in W**". This instruction places a literal value in the *W* register, and then pops the return address from the stack, to return execution to the calling code.

Note that the baseline PICs do not have a simple 'return' instruction, only 'retlw'; you can't avoid returning a literal in *W*. If you need to preserve the value in *W* when a subroutine is called, you must first save it in another register.

Here is the 500 ms delay routine, written as a subroutine:

```
delay500                ; delay 500ms
    movlw    .244        ; outer loop: 244x(1023+1023+3)-1+3+4
    movwf    dc2        ;   = 499,962 cycles
    clrf     dc1
dly1    nop            ; inner loop 1 = 256x4-1 = 1023 cycles
    decfsz  dc1,f
    goto    dly1
dly2    nop            ; inner loop 2 = 1023 cycles
    decfsz  dc1,f
    goto    dly2
    decfsz  dc2,f
    goto    dly1

    retlw   0
```

Note that this code returns a '0' in *W*. It doesn't have to be '0'; any number would do, but it's conventional to return a '0' if you're not returning some specific value.

Parameter Passing with W

A re-usable 500 ms delay routine is all very well, but it's only useful if you need a delay of 500 ms. What if you want a 200 ms delay – write another routine? Have multiple delay subroutines, one for each delay length? It's more useful to have a single routine that can provide a range of delays. The requested delay time would be passed as a *parameter* to the delay subroutine.

If you had a number of parameters to pass (for example, a 'multiply' subroutine would have to be given the two numbers to multiply), you'd need to place the parameters in general purpose registers, accessed by both the calling program and the subroutine. But if there is only one parameter to pass, it's often convenient to simply place it in *W*.

For example, in the delay routine above, we could simply remove the 'movlw .244' line, and instead pass this number (244) as a parameter:

```
    movlw    .244
    call    delay        ; delay 244 x 2.049ms = 500ms
```

But passing a value of '244' to specify a delay of 500 ms is a little obscure. It would be better if the delay subroutine worked in multiples of an easier-to-use duration than 2.049 ms.

Ideally, we'd pass the number of milliseconds wanted, directly, i.e. pass a parameter of '500' for a 500 ms delay. But that won't work. The baseline PICs are 8-bit devices; the largest value you can pass in any single register, including *W*, is 255.

If the delay routine produces a delay which is some multiple of 10 ms, it could be used for any delay from 10 ms to 2.55 s, which is quite useful – you'll find that you commonly want delays in this range.

To implement a $W \times 10$ ms delay, we need an inner set of loops which create a 10 ms (or close enough) delay, and an outer loop which counts the specified number of those 10 ms loops.

To count multiples of 10 ms, we need to add a third loop counter, as in the following code:

```

delay10                ; delay W x 10ms
    movwf    dc3        ; delay = 1+Wx(3+10009+3)-1+4 -> Wx10.015ms

dly2    movlw    .13    ; repeat inner loop 13 times
        movwf    dc2    ; -> 13x(767+3)-1 = 10009 cycles

        clrf     dc1    ; inner loop = 256x3-1 = 767 cycles
dly1    decfsz   dc1,f
        goto    dly1

        decfsz   dc2,f  ; end middle loop
        goto    dly1

        decfsz   dc3,f  ; end outer loop
        goto    dly2

    retlw    0

```

Example 1: Flash LED (using delay subroutine with parameter passing)

To illustrate where subroutines and parameter passing are useful, suppose that, instead of the LED being on half the time (a 50% *duty cycle*), we want the LED to flash briefly, for say 200 ms, once per second (a 20% duty cycle).

That would require a delay of 200 ms while the LED is on, then a delay of 800 ms while it is off.

We'll demonstrate this using the circuit shown on the right.

It's the same as the circuit used in the last two lessons, except that we're now using a 12F509 instead of a 10F200 or 12F508.

If you have a Gooligum training board, you should remove the PIC10F200 from the '10F' socket, and instead plug a PIC12F509 into the top section of the 14-pin IC socket – the section marked '12F'³. And as before, connect jumper JP12, to enable the LED on GP1.

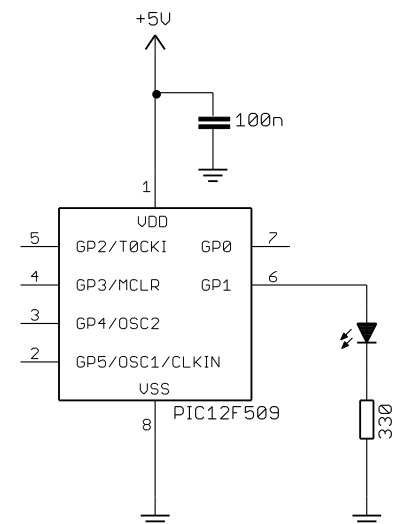
If you have the Microchip Low Pin Count Demo board, refer back to [lesson 1](#) to see how to build this circuit, either by soldering a resistor and to the demo board, or by making connections on the demo board's 14-pin header.

The first part of our program will be much the same as before, except that we need to change the processor identification section, to reflect the fact that we're now using a 12F509:

```

list          p=12F509
#include      <p12F509.inc>

```



³ Note that, although the PIC12F509 comes in an 8-pin package, **it will not work** in the 8-pin '10F' socket. You must install it in the '12F' section of the 14-pin socket.

The configuration section is the same as in our previous 12F508 code:

```

; ext reset, no code protect, no watchdog, int RC clock
__CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC

```

However, the memory address of the RC calibration section has to be changed; the 12F509 has more memory, extending from 000h to 3FFh, so its calibration instruction is at 0x3FF:

```

RCCAL    CODE    0x3FF        ; processor reset vector
         res 1        ; holds internal RC cal value, as a movlw k

```

Using the ‘delay10’ subroutine presented above, our main loop becomes:

```

main_loop
; turn on LED
movlw   b'000010'        ; set GP1 (bit 1)
movwf   GPIO
; delay 0.2 s
movlw   .20              ; delay 20 x 10 ms = 200 ms
call    delay10
; turn off LED
clrf   GPIO              ; (clearing GPIO clears GP1)
; delay 0.8 s
movlw   .80              ; delay 80 x 10ms = 800ms
call    delay10

; repeat forever
goto   main_loop

```

Note that this code does not use a shadow register. It’s no longer necessary, because the GP1 bit is being directly set/cleared. It’s not being flipped; there’s no dependency on its previous value. At no time does the GPIO register have to be read. It’s only being written to. So “read-modify-write” is not a consideration here. If that’s unclear, go back to the description in [lesson 2](#), and think about why an ‘xor’ operation on an I/O register is different to simply writing a new value directly to the I/O register. It’s important to understand this point, but if you’re ever in doubt about whether the “read-modify-write” problem may apply, it’s best to be safe and use a shadow register.

Complete program

Here is the complete program to do this, illustrating how all the above pieces fit together.

You’ll see that the subroutine has been placed into a “SUBROUTINES” section toward the end, and clearly documented – if you’re using subroutines in your code, it’s good to be able to easily find them and see what they do, in case you’ve forgotten, or if you want to re-use a subroutine in another program:

```

;*****
;
; Description:    Lesson 3, example 1
;
; Demonstrates simple subroutine calls with parameter passing
;
; Flashes a LED at approx 1 Hz, with 20% duty cycle
; LED continues to flash until power is removed
;
;*****
;
; Pin assignments:
;   GP1 = flashing LED
;
;*****

```

```

list      p=12F509
#include   <p12F509.inc>

;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, int RC clock
__CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntrC_OSC

;***** VARIABLE DEFINITIONS
                UDATA
dc1         res 1                ; delay loop counters
dc2         res 1
dc3         res 1

;***** RC CALIBRATION
RCCAL       CODE    0x3FF        ; processor reset vector
                res 1            ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET       CODE    0x000        ; effective reset vector
                movwf   OSCCAL    ; apply internal RC factory calibration

;***** MAIN PROGRAM *****

;***** Initialisation
start
                movlw   b'111101'    ; configure GP1 (only) as an output
                tris    GPIO

;***** Main loop
main_loop
                ; turn on LED
                movlw   b'000010'    ; set GP1 (bit 1)
                movwf   GPIO
                ; delay 0.2 s
                movlw   .20           ; delay 20 x 10 ms = 200 ms
                call    delay10
                ; turn off LED
                clrf    GPIO          ; (clearing GPIO clears GP1)
                ; delay 0.8 s
                movlw   .80           ; delay 80 x 10ms = 800ms
                call    delay10

                ; repeat forever
                goto    main_loop

;***** SUBROUTINES *****

;***** Variable delay: 10 ms to 2.55 s
;
; Delay = W x 10 ms
;
delay10
                movwf   dc3            ; delay = 1+Wx(3+10009+3)-1+4 = W x 10.015ms
dly2         movlw   .13            ; repeat inner loop 13 times

```

```

        movwf    dc2                ; -> 13x(767+3)-1 = 10009 cycles
        clrf    dc1                ; inner loop = 256x3-1 = 767 cycles
dly1    decfsz  dc1,f
        goto    dly1
        decfsz  dc2,f                ; end middle loop
        goto    dly1
        decfsz  dc3,f                ; end outer loop
        goto    dly2

        retlw   0

        END

```

CALL Instruction Address Limitation

Before moving on from subroutines, it is important that you be aware of another limitation in the baseline PIC architecture, this time regarding the addressing of subroutines.

We saw above that the opcode for the `goto` instruction is only three bits long, with the remaining nine bits in the 12-bit instruction giving the address to jump to.

However, the opcode for the `call` instruction is 1001. That's four bits, leaving only eight bits to specify the address of the subroutine being called.

Eight bits can hold a value from 0 to 255. There's no problem if your subroutine is in the first 256 words of a memory page. But what if it's at an address above 255? With only eight bits available for the address in the `call` instruction, how can you specify an address higher than 255? The answer is that, on the baseline PICs, you can't.

In the baseline PIC architecture, subroutine calls are limited to the first 256 locations of any program memory page.

Although it's possible for a subroutine to use `goto` to jump to anywhere in a memory, the entry point for every subroutine must be within the first 256 words of a memory page.

That can be an awkward limitation to work around; if your main code is more than 256 instructions long and (as in the program above) you place your subroutines immediately after the main code, you'll have a problem.

The MPASM assembler will warn you if you try to call a subroutine past the 256-word boundary, but the only way to fix it is to re-arrange your code.

One approach would be to place the subroutines toward the beginning of the main code section, which we know is located at address 0x000 (the start of the first page), with a `goto` instruction immediately before the subroutines, to jump around them to the start of the main program code. A problem with that approach is that all the subroutines plus the main code may be too big to fit into a single page (i.e. more than 512 words in total), but any one code section has to fit within a single page.

The solution to that is simple – place the subroutines in the section located at 0x000 (so we know they are toward the start of a page), but put the main code into its own code section, which the linker can place anywhere in program memory – wherever it fits – perhaps on a different page.

However this doesn't necessarily mean that the subroutines will all start within the first 256 words in the page; if the subroutines together total more than 256 instruction words, there could still be problems.

A robust solution is to use a jump table, or *subroutine vectors* (or *long calls*). The idea is that only the entry points for each subroutine are placed at the start of a page. Each entry point consists of a '`goto`' instruction,

jumping to the main body of the subroutine, which could be anywhere in memory – preferably in another CODE section so that the linker is free to place it wherever it fits best.

The previous program could be restructured to use a subroutine vector, as follows:

```

;***** RESET VECTOR *****
RESET    CODE    0x000          ; effective reset vector
          movwf   OSCCAL        ; apply internal RC factory calibration
          goto    start         ; jump to main code

;***** Subroutine vectors
delay10  goto    delay10_R     ; delay W x 10ms

;***** MAIN PROGRAM *****
MAIN     CODE

;***** Initialisation
start
          movlw   b'111101'     ; configure GP1 (only) as an output
          tris    GPIO

;***** Main loop
main_loop
          ; turn on LED
          movlw   b'000010'     ; set GP1 (bit 1)
          movwf   GPIO
          ; delay 0.2 s
          movlw   .20           ; delay 20 x 10 ms = 200 ms
          call    delay10
          ; turn off LED
          clrf    GPIO         ; (clearing GPIO clears GP1)
          ; delay 0.8 s
          movlw   .80           ; delay 80 x 10ms = 800ms
          call    delay10

          goto    main_loop     ; repeat forever

;***** SUBROUTINES *****
SUBS     CODE

;***** Variable delay: 10 ms to 2.55 s
;
; Delay = W x 10 ms
;
delay10_R
          movwf   dc3           ; delay = 1+Wx(3+10009+3)-1+4 = W x 10.015ms
dly2     movlw   .13           ; repeat inner loop 13 times
          movwf   dc2           ; -> 13x(767+3)-1 = 10009 cycles
          clrf    dc1           ; inner loop = 256x3-1 = 767 cycles
dly1     decfsz  dc1,f
          goto    dly1
          decfsz  dc2,f         ; end middle loop
          goto    dly1
          decfsz  dc3,f         ; end outer loop
          goto    dly2

          retlw   0

```

Dividing the program into so many CODE sections is overkill for such a small program, but if you adopt this approach you will avoid potential problems as your programs grow larger.

The entry point for the ‘delay10’ subroutine is guaranteed to be within the first 256 words of the program memory page, while the subroutine proper, renamed to ‘delay10_R’ (R for routine) is in a separate code section which could be anywhere in memory – perhaps on a separate page.

And therein lies a problem; as written, this code is not guaranteed to work! As explained earlier, a `goto` or `call` only works correctly if the address you are jumping to is in the same page as the address you are jumping from – unless you have set the page selection bits correctly first.

Using the *PAGESEL* directive

If your program includes multiple code sections, you can’t know beforehand where the linker will place them in memory, so you can’t know how to set the page selection bits when jumping to or calling locations in other sections.

The solution is to use the ‘`pagesel`’ directive, which instructs the assembler and linker to generate code to select the correct page for the given program address.

To ensure that the program above will work correctly, regardless of which pages the main code and subroutines are on, `pagesel` directives should be added to the start-up and subroutine vector code, as follows:

```
RESET    CODE    0x000                ; effective reset vector
         movwf   OSCCAL                ; apply internal RC factory calibration
         pagesel start
         goto    start                ; jump to main code

;***** Subroutine vectors
delay10                ; delay W x 10ms
         pagesel delay10_R
         goto    delay10_R
```

And, then, since the `delay10` subroutine entry point may be in a different page from the main code, `pagesel` directives should be added to the main loop, as follows:

```
main_loop
; turn on LED
movlw   b'000010'      ; set GP1 (bit 1)
movwf   GPIO
; delay 0.2 s
movlw   .20            ; delay 20 x 10 ms = 200 ms
pagesel delay10
call    delay10
; turn off LED
clrf   GPIO            ; (clearing GPIO clears GP1)
; delay 0.8 s
movlw   .80            ; delay 80 x 10ms = 800ms
call    delay10

pagesel main_loop     ; repeat forever
goto    main_loop
```

Note that there is no ‘`pagesel`’ before the second call to ‘`delay10`’. It’s unnecessary, because the first ‘`pagesel`’ has already set the page selection bits for calls to ‘`delay10`’. If you’re going to successively call subroutines in a single section, there is no need to add a ‘`pagesel`’ for each; the first is enough.

Finally, note the ‘`pagesel`’ before the ‘`goto`’ at the end of the loop. This is necessary because, at that point, the page selection bits will still be set for whatever page the ‘`delay10`’ entry point is on, not necessarily the current page.

An alternative is to place a `pagesel $` directive (“select page for current address”) after each `call` instruction, to ensure that the current page is selected after returning from a subroutine.

You do not, however, need to use `pagesel` before every `goto` or `call`, or after every `call`. Remember that a single code section is guaranteed to be wholly contained within a single page⁴. So, once you know that you’ve selected the correct page, subsequent `goto` or `call` instructions to addresses in the same section will work correctly. But be careful!

If in doubt, using `pagesel` before every `goto` and `call` is a safe approach that will always work.

Example 2: Flash LED (calling subroutine via jump table)

To clearly show how subroutine vectors and the `pagesel` directive are used, here are the reset, main and subroutine code sections of our “flash an LED with a 20% duty cycle” program:

```

;***** RESET VECTOR *****
RESET    CODE    0x000                ; effective reset vector
         movwf   OSCCAL              ; apply internal RC factory calibration
         pagesel start
         goto    start                ; jump to main code

;***** Subroutine vectors
delay10                      ; delay W x 10ms
         pagesel delay10_R
         goto    delay10_R

;***** MAIN PROGRAM *****
MAIN     CODE

;***** Initialisation
start
         movlw   b'111101'           ; configure GP1 (only) as an output
         tris    GPIO

;***** Main loop
main_loop
         ; turn on LED
         movlw   b'000010'           ; set GP1 (bit 1)
         movwf   GPIO
         ; delay 0.2 s
         movlw   .20                  ; delay 20 x 10 ms = 200 ms
         pagesel delay10
         call    delay10
         ; turn off LED
         clrf    GPIO                 ; (clearing GPIO clears GP1)
         ; delay 0.8 s
         movlw   .80                  ; delay 80 x 10ms = 800ms
         call    delay10

         ; repeat forever
         pagesel main_loop
         goto    main_loop

;***** SUBROUTINES *****
SUBS     CODE

```

⁴ unless you are an advanced PIC developer and create your own linker scripts...

```

;***** Variable delay: 10 ms to 2.55 s
;
; Delay = W x 10 ms
;
delay10_R
    movwf    dc3                ; delay = ?+1+Wx(3+10009+3)-1+4 = W x 10.015ms
dly2     movlw    .13           ; repeat inner loop 13 times
        movwf    dc2           ; -> 13x(767+3)-1 = 10009 cycles
        clrf     dc1           ; inner loop = 256x3-1 = 767 cycles
dly1     decfsz   dc1,f         ;
        goto    dly1
        decfsz   dc2,f         ; end middle loop
        goto    dly1
        decfsz   dc3,f         ; end outer loop
        goto    dly2

    retlw    0

```

Relocatable Modules

If you wanted to take a subroutine you had written as part of one program, and re-use it in another, you could simply copy and paste the source code into the new program.

There are a few potential problems with this approach:

- Address labels, such as ‘dly1’, may already be in use in the new program or in other pieces of code that you’re copying.
- You need to know which variables are needed by the subroutine, and remember to copy their definitions to the new program.
- Variable names have the same problem as address labels – they may already be used in new program, in which case you’d need to identify and rename all references to them.

These problems can be avoided by keeping the subroutine code in a separate source file, where it is assembled into an object file. The main code is assembled into a separate object file. These object files – one for the main code, plus one for each module, are then linked together to create the final executable code, which is output as a .hex file to be programmed into the PIC. This assembly/link (or *build*) process sounds complicated, but MPLAB takes care of the details, as we’ll see later.

To be relocatable, a module must have its own code sections, which the linker can place anywhere in memory (hence the term ‘relocatable’).

It must also have its own data sections, to keep its variables separate from the rest of the program’s variables. Again, the linker can place these data sections anywhere in data memory – perhaps in a different bank from your other variables.

When you are using more than one data section, which will usually be the case if you are using relocatable modules, you must ensure that you set the bank selection bits correctly when accessing variables.

Using the *BANKSEL* directive

Typically, when you use the `UDATA` and `RES` directives to declare and allocate space for variables, you don’t specify an address, allowing the linker to locate the section anywhere in data memory, fitting it around other sections. The potential problem with this is that “anywhere in data memory” also means “in any bank”.

When you refer to registers allocated within relocatable data sections, you can’t know what bank they will be in, so you can’t know how to set the bank selection bits in `FSR`.

The solution is similar to that for paging: use the `banksel` directive to instruct the assembler and linker to generate appropriate code to select the correct bank for the given variable (or data address label).

To ensure that the ‘delay10’ routine accesses the register bank containing the delay loop counter variables, a `banksel` directive should be added, as follows:

```

delay10_R
    banksel dc3                ; delay = ?+1+Wx(3+10009+3)-1+4 = W x 10.015ms
    movwf dc3
dly2   movlw .13              ; repeat inner loop 13 times
    movwf dc2                ; -> 13x(767+3)-1 = 10009 cycles
    clrf dc1                 ; inner loop = 256x3-1 = 767 cycles
dly1   decfsz dc1,f
    goto dly1
    decfsz dc2,f             ; end middle loop
    goto dly1
    decfsz dc3,f             ; end outer loop
    goto dly2

    retlw 0

```

`banksel` is used the first time a group of variables is accessed, but not subsequently – unless another bank has been selected (for example, after calling a subroutine which may have selected a different bank).

We know that all three variables will be in the same bank, since they are all declared as part of the same data section⁵. If you select the bank for one variable in a data section, then it will also be the correct bank for every other variable in that section, so we only need to use `banksel` once. You only need another `banksel` if you’re going to access a variable in a different data section.

Note that the code could have been started with ‘`banksel dc1`’, instead of ‘`banksel dc3`’; it would make no difference, because `dc1` and `dc3` are in the same section and therefore the same bank. But it seems clearer, and more maintainable, to have `banksel` refer to the variable you’re about to access, and to place it immediately before that access.

Declaring a Shared Data Section

As discussed earlier, not all data memory is banked. The special function registers and some of the general purpose registers are mapped into every bank. These shared registers are useful for storing variables that are used throughout a program, without having to worry about setting bank selection bits to access them.

The `UDATA_SHR` directive is used to declare a section of shared data memory.

It’s used in the same way as `UDATA`; the only difference is that registers reserved in a `UDATA_SHR` section won’t be banked.

Since there is less shared memory available than banked memory, it should be used sparingly. However, it can make sense to allocate shadow registers in shared memory, as they are likely to be used often.

To summarise:

- The first time you access a variable declared in a `UDATA` section, use `banksel`.
- To access subsequent variables in the same `UDATA` section, you don’t need to use `banksel`. (unless you had selected another bank between variable accesses)
- Following a call to a subroutine or external module, which may have selected a different bank, use `banksel` for the first variable accessed after the call.
- To access variables in a `UDATA_SHR` section, there is never any need to use `banksel`.

⁵ again assuming that you’re not an advanced developer with custom linker scripts...

Example 3: Flash LED (using a relocatable module)

To demonstrate how to use re-usable code modules, we'll take our general-purpose delay subroutine, and place it in a separate file. We'll then call this *external module* from the main program.

We'll setup a project with the following files:

- delay10.asm - containing the $W \times 10$ ms delay routine
- BA_L3-Flash_LED-main.asm - the main code (calling the delay routine)

(or whatever names you choose)

How to do this depends on whether you're using MPLAB 8 or MPLAB X, so again we'll look at both.

Creating a multiple-file project, using MPLAB 8.xx

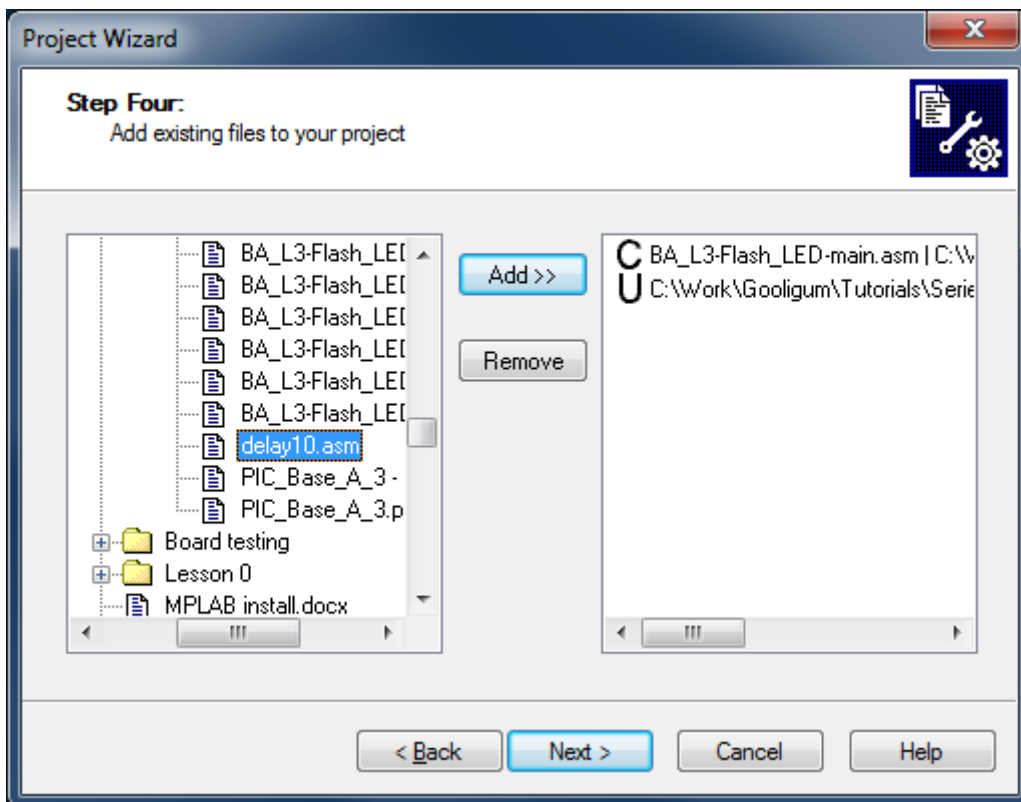
To create the multiple-file project, open an existing project and then save it with a new name, such as "BA_L3-Flash_LED-mod", in the same way as you did when creating a new project in [lesson 2](#).

Open the assembler (.asm) source file from example 2, containing the main loop and the 'delay10' subroutine, and save it, using "File → Save As..." as "delay10.asm".

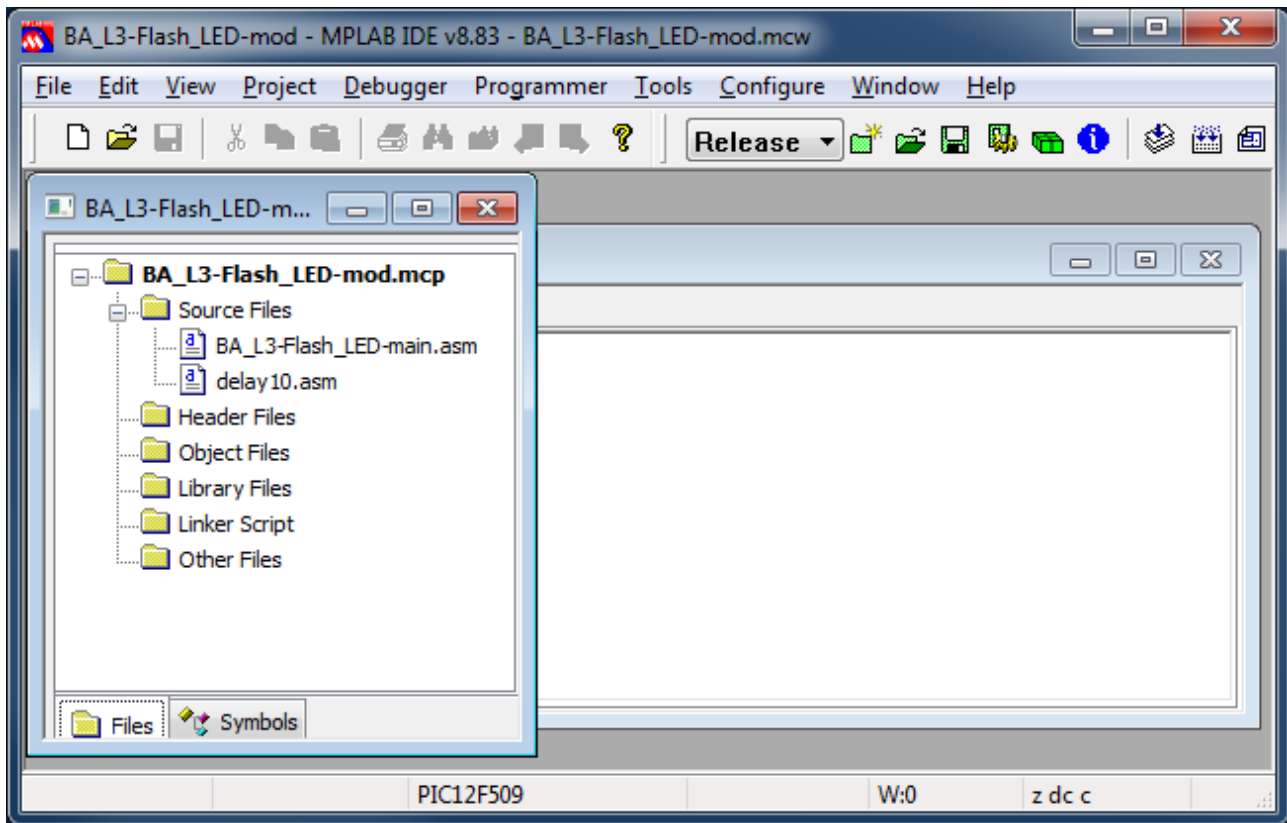
Next close the editor window and run the project wizard to reconfigure the active project, as before.

When you reach "Step Four: Add existing files to your project" window, rename the source file to "BA_L3-Flash_LED-main.asm" (for example), in the same way as was done in lesson 2 – changing the "U" next to the filename to "C", and editing the file name.

Now find the "delay10.asm" file you saved before in the left hand pane, and click on "Add>>" to add it to your project. The filename is already correct, but you should click on the "A" next to the filename to change it to a "U" to indicate that this is a user file, as shown:



After clicking “Next >” and then “Finish”, you will see that your project now contains both source files:

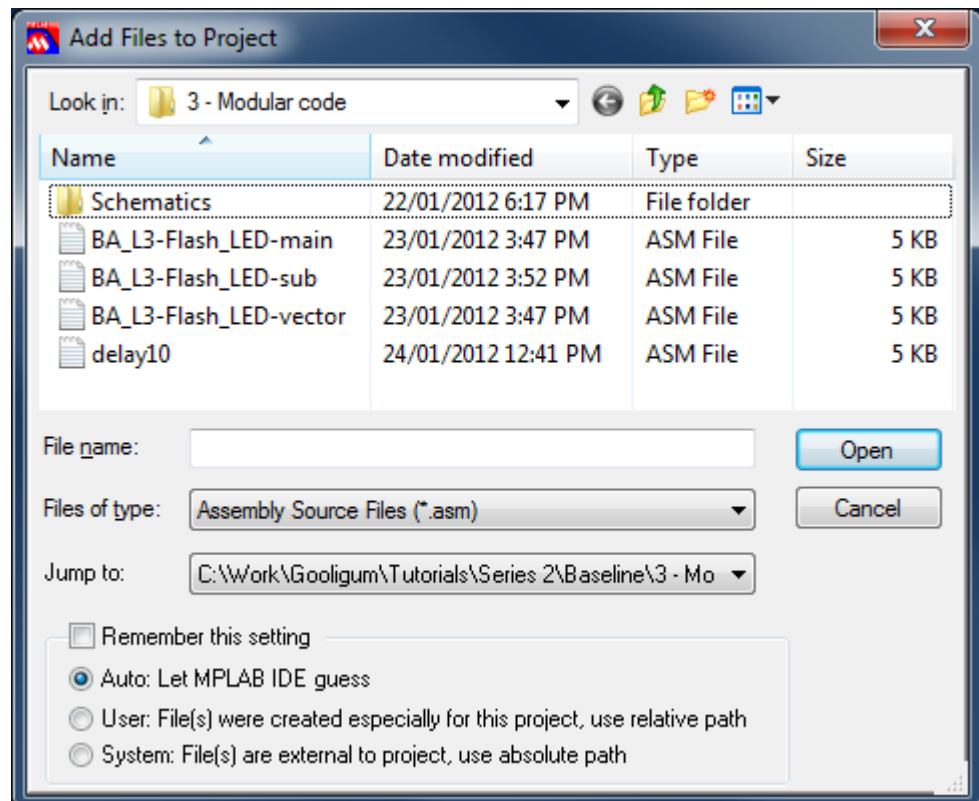


Of course there are a number of ways to create a multiple-file project.

If you simply want to add an existing file (or files) to a project, you can right-click on “Source Files” in the project window, and then select “Add Files” from the context menu, or else select the “Project → Add Files to Project...” menu

item. Either way, you will be presented with the window shown on the right. As you can see, it gives you the option, for each file, to specify whether it is a user (relative path) or system (absolute path) file. If you’re unsure, just select “Auto” and let MPLAB decide.

If you want to create a new file from scratch, instead of using an existing one, use the “Project → Add New File to Project...” menu item (also available



under the File menu). You'll be presented with a blank editor window, into which you can copy text from other files (or simply start typing!).

Creating a multiple-file project, using MPLAB X

To create the multiple-file project, open an existing project and then save it with a new name, such as "BA_L3-Flash_LED-mod", in the same way as you did when creating new project in [lesson 2](#).

Rename the source file to "BA_L3-Flash_LED-main.asm" (for example), in the same way as was done in lesson 2 – right-click it in the Projects window and select "Rename...".

Next we need to copy this file, creating a new file which will contain our delay module.

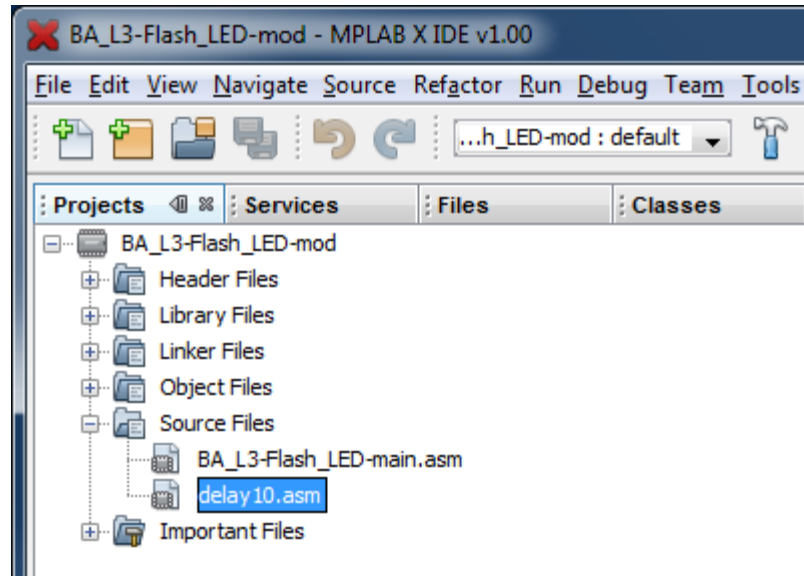
There are a few ways to do this, but the easiest is probably to right-click the source file in the Projects window and select "Copy".

Right-click "Source Files" in the project tree, and select "Paste".

A new .asm file (a copy of the original) should appear in the project tree.

You can now right-click this new file, and rename it to "delay10.asm".

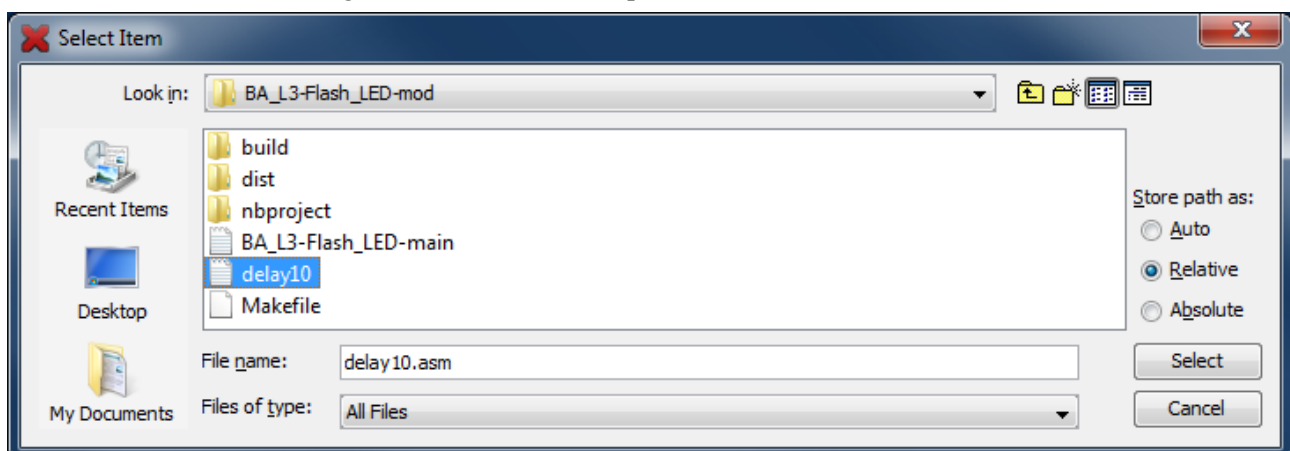
Your project should look like the one shown on the right.



Another way to do this is to double-click the original source file (the one you want to copy), opening an editor window. If you now activate the editor window, by clicking anywhere in it, you can use the "File → Save As..." menu item to save the file as "delay10.asm".

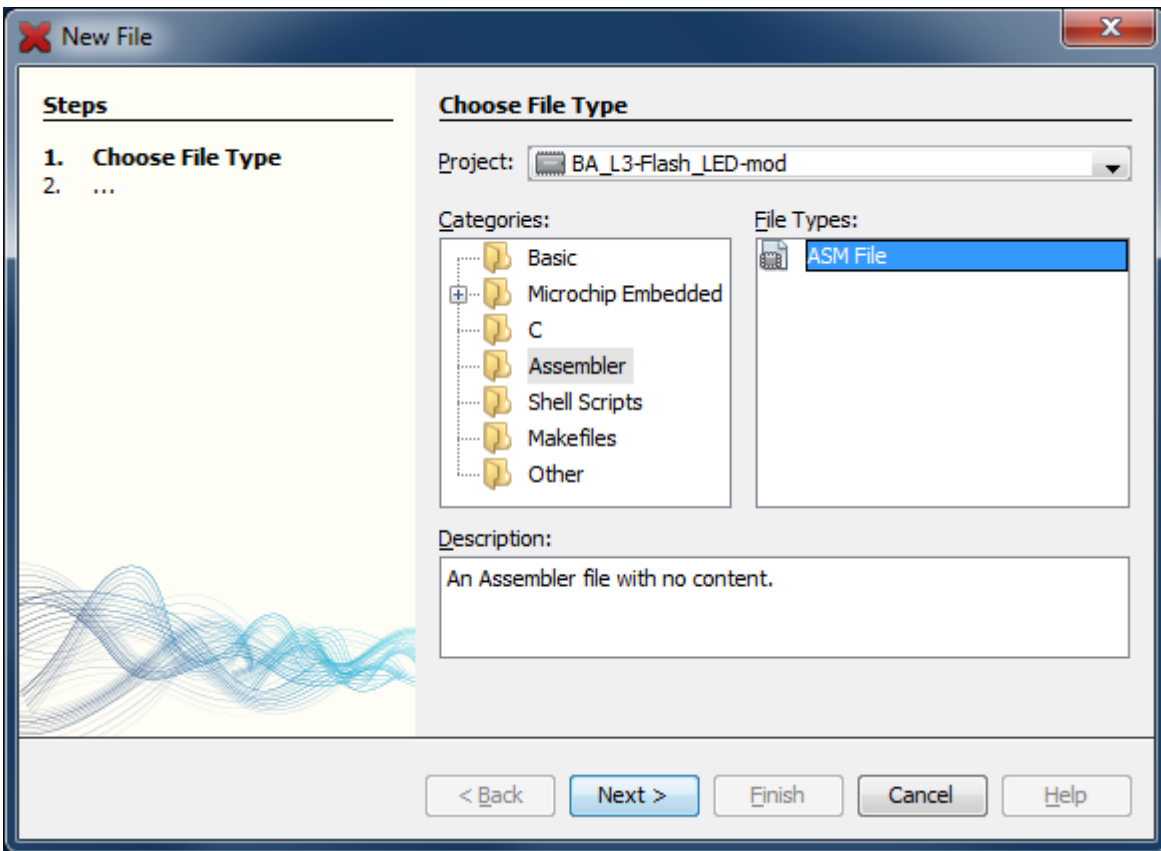
The only problem is that this new source file hasn't appeared in the Projects window; MPLAB X doesn't yet know that the new file is part of the project. So, we need to add it.

To add an existing file (or files) to a project, you can right-click on "Source Files" in the Projects window, and then select "Add Existing Item...". You will be presented with the window shown below:

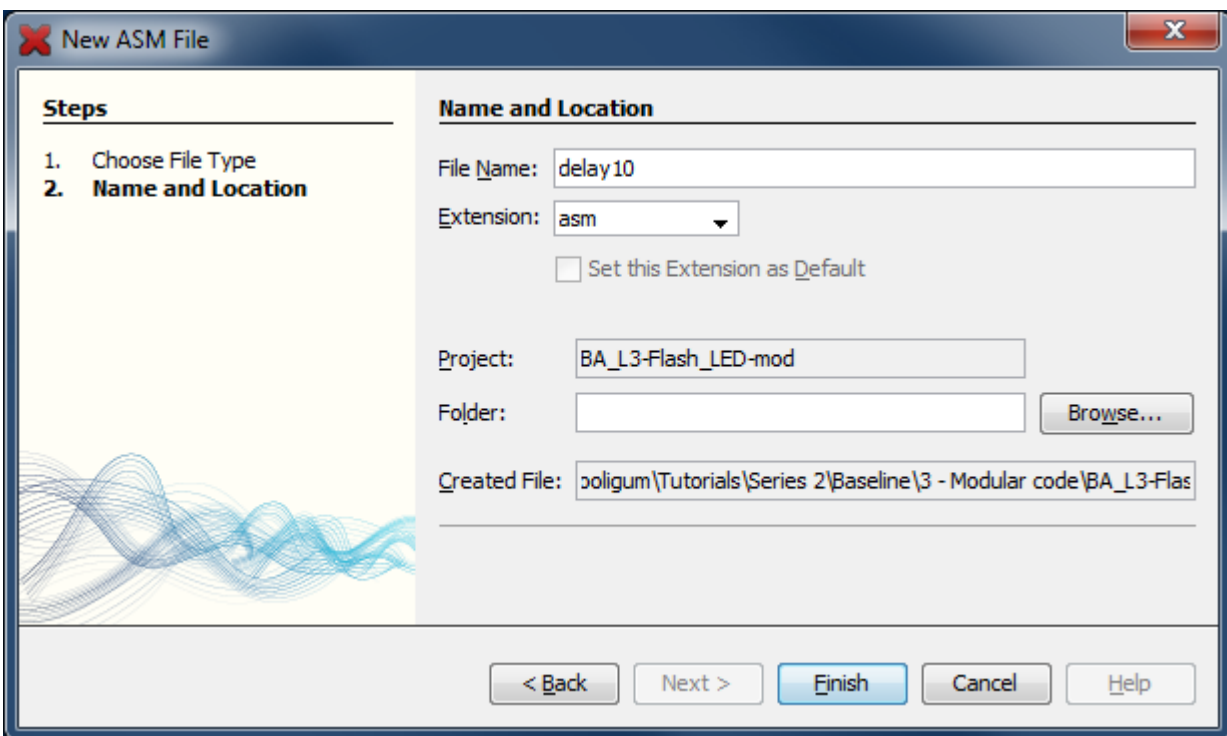


As you can see, it gives you the option to specify whether the file has a relative path (appropriate for most "user" files) or absolute path (for most "system" files). If you're unsure, just select "Auto" and let MPLAB decide.

If you want to create a new file from scratch, instead of using an existing one, you can use the “File → New File...” menu item, in which case you’ll be asked to choose the file type. You should select “Assembler” from the Categories window, and the “ASM File” file type, and then click “Next >”:



You’ll be presented with the “New ASM File” window, which you can also get to (more easily) by right-clicking your project in the Projects window, and selecting “New → ASM File...”:



When you click “Finish”, the new file will be appear in the project tree, and you will be presented with a blank editor window, into which you can copy text, such as the delay subroutine, from other files (or simply start typing!).

However you created them, now that you have a project which includes the two source files, we can consider their content...

Creating a relocatable module

Converting an existing subroutine, such as our ‘delay10’ routine, into a standalone, relocatable module is easy. All you need to do is to declare any symbols (address labels or variables) that need to be accessible from other modules, using the GLOBAL directive.

Here is the complete “delay10.asm” file:

```

;*****
;
; Architecture: Baseline PIC
; Processor: any
;
;*****
;
; Files required: none
;
;*****
;
; Description: Variable Delay : N x 10 ms (10 ms - 2.55 s)
;
; N passed as parameter in W reg
; exact delay = W x 10.015ms
;
; Returns: W = 0
; Assumes: 4 MHz clock
;
;*****

#include <p12F509.inc> ; any baseline device will do

GLOBAL delay10_R

;***** VARIABLE DEFINITIONS
        UDATA
dc1     res 1           ; delay loop counters
dc2     res 1
dc3     res 1

;***** SUBROUTINES *****
        CODE

;***** Variable delay: 10 ms to 2.55 s
;
; Delay = W x 10 ms
;
delay10_R
        banksel dc3           ; delay = ?+1+Wx(3+10009+3)-1+4 = W x 10.015 ms
        movwf dc3

```



```

dly2    movlw    .13                ; repeat inner loop 13 times
        movwf   dc2                ; -> 13x(767+3)-1 = 10009 cycles
        clrf    dc1                ; inner loop = 256x3-1 = 767 cycles
dly1    decfsz  dc1,f              ;
        goto    dly1
        decfsz  dc2,f              ; end middle loop
        goto    dly1
        decfsz  dc3,f              ; end outer loop
        goto    dly2

        retlw   0

        END

```

This consists of the subroutine from the earlier example, plus a `UDATA` section to reserve data memory for its variables. Because this memory is banked, a `banksel` directive has been added to ensure that the bank containing these variables is accessed.

Toward the start, a `GLOBAL` directive has been added to declare that the `'delay10_R'` label is to be made available (*exported*) to other modules, allowing them to call this subroutine.

You should also include (pardon the pun) a `#include` directive, to define any “standard” symbols used in the code, such as the instruction destinations `'w'` and `'f'`. This delay routine will work on any baseline PIC; it's not specific to any, so you can use the include file for any of the baseline PICs, such as the 12F509.

Note that there is no `list` directive; this avoids the processor mismatch errors that would be reported if you specify more than one processor in the modules comprising a single project.

Of course it's also important to add a block of comments at the start; they should describe what this module is for, how it is used, any effects it has (including side effects, such as returning `'0'` in the `W` register), and any assumptions that have been made. In this case, this routine will generate the expected delay if the processor is clocked at exactly 4 MHz. This assumption should be documented in the comments.

Calling relocatable modules

Having created an *external* relocatable module (i.e. one in a separate file), we need to declare, in the main (or *calling*) file any labels we want to use from the module being called, so that the linker knows that these labels are defined in another module. That's done with the `EXTERN` directive.

Here is the complete example “main code” file (“`BA_L3-Flash_LED-main.asm`”), which calls the delay module:

```

;*****
;
; Architecture: Baseline PIC
; Processor:    12F508/509
;
;*****
;
; Files required: delay10.asm      (provides W x 10 ms delay)
;
;*****
;
; Description:    Lesson 3, example 3
;
; Demonstrates how to call external modules
;
; Flashes a LED at approx 1 Hz
; LED continues to flash until power is removed
;
;*****

```

```

;                                                                 *
; Pin assignments:                                             *
;   GP1 = flashing LED                                       *
;                                                                 *
;*****
list      p=12F509
#include  <p12F509.inc>

EXTERN   delay10_R      ; W x 10 ms delay

;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, int RC clock
__CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC

;***** VARIABLE DEFINITIONS
                UDATA_SHR
sGPIO      res      1          ; shadow copy of GPIO

;***** RC CALIBRATION
RCCAL      CODE     0x3FF      ; processor reset vector
                res 1          ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET      CODE     0x000      ; effective reset vector
                movwf   OSCCAL   ; apply internal RC factory calibration
                pagesel start
                goto    start   ; jump to main code

;***** Subroutine vectors
delay10    ; delay W x 10 ms
                pagesel delay10_R
                goto    delay10_R

;***** MAIN PROGRAM *****
MAIN       CODE

;***** Initialisation
start
                movlw   b'111101' ; configure GP1 (only) as an output
                tris   GPIO

                clrf   sGPIO      ; start with shadow GPIO zeroed

;***** Main loop
main_loop
                ; toggle LED on GP1
                movf   sGPIO,w    ; get shadow copy of GPIO
                xorlw  b'000010'  ; toggle bit corresponding to GP1 (bit 1)
                movwf  sGPIO      ; in shadow register
                movwf  GPIO       ; and write to GPIO
                ; delay 0.5 s
                movlw  .50        ; delay 50 x 10 ms = 500 ms
                pagesel delay10   ; -> 1 Hz flashing at 50% duty cycle
                call   delay10

```

```

; repeat forever
pagesel main_loop
goto    main_loop

```

```

END

```

Instead of re-using the main code from the previous example, this is actually an adaptation of the “Flash an LED” program from [lesson 2](#), because that program used a shadow register – allowing us to demonstrate that the main program can have its own variables, in their own data section, with no need to declare or reference the external module’s variables at all.

The shadow register is declared as a shared (non-banked) variable by placing it in a `UDATA_SHR` section, so there is no need to use `banksel` before accessing it.

The inline delay routine has been replaced with a call our external delay module, and the variables used by the delay routine removed. And toward the start of the program, an `EXTERN` directive has been added, to declare that the ‘`delay10_R`’ label is a reference to another module.

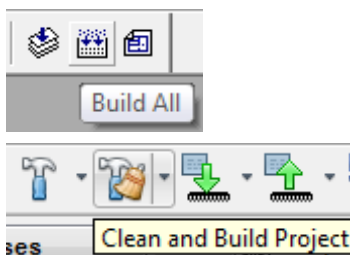
Note that a subroutine vector is still used (to avoid potential problems due to the baseline architecture’s subroutine addressing limitation, explained earlier), as it is not possible to know where in program memory the linker will place the module.

You should also document, in the comments block at the start of the source code, the fact that this program relies on an external module, what that module does, and what file it is defined in.

To summarise:

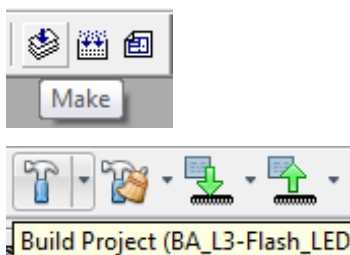
- The `GLOBAL` and `EXTERN` directives work as a pair.
- `GLOBAL` is used in the file that defines a module, to export a symbol for use by other modules.
- `EXTERN` is used when calling external modules. It declares that a symbol has been defined elsewhere.

The Build Process (Revisited)



In a multiple-file project, when you select “Project → Build All” or click on the “Build All” toolbar button (in MPLAB 8), or select “Run → Clean and Build” or click on the “Clean and Build” toolbar button (in MPLAB X), the assembler will assemble all the source files, producing a new ‘.o’ *object file* for each. The linker then combines these ‘.o’ files to build a single ‘.hex’ file, containing an image of the executable code to be programmed into the PIC.

If, however, you’ve been developing a multi-file project, and you’ve already built it, and then go back and alter just one of the source files, there’s no need to re-assemble all the other source files, if they haven’t changed. The object files corresponding to those unchanged source files will still be there, and they’ll still be valid.



That’s what the “Project → Make” menu item or the “Make” toolbar button (in MPLAB 8), or “Run → Build” or the “Build” toolbar button (in MPLAB X) do, as was discussed briefly in [lesson 1](#). Like “Build All” or “Clean and Build”, it builds your project, but it only assembles source files which have a newer date stamp than the corresponding object file. This is what you

normally want, to save unnecessary assembly time (not that it makes much difference with such a small project!), so MPLAB 8 includes a handy shortcut for “Make” – just press ‘F10’. And as we saw in lesson 1, MPLAB X goes a step further, providing a single toolbar button to “Make and Program Device” – or just press ‘F6’.

After you build (or make) the project, you’ll see a number of new files in the project directory⁶. In addition to your ‘.asm’ source files and the ‘.o’ object files and the ‘.hex’ output file we’ve already discussed, you’ll find ‘.lst’ files corresponding to each of the source files, and a ‘.map’ file corresponding to the project name⁷.

I won’t describe these in detail, but they are worth looking at if you are curious about the build process. And they can be valuable to refer to if you when debugging, as they show exactly what the assembler and linker are doing.

The ‘.lst’ *list files* show the output of the assembler; you can see the opcodes corresponding to each instruction. They also show the value of every label. But you’ll see that, for the list files belonging to the source files (e.g. ‘delay10.lst’), they contain a large number of question marks. For example:

```
0000          00050 delay10_R
0000  ????? 00051      banksel dc3          ; delay = ?+1+Wx(3+10009+3)-1+4 = W x 10.015 ms
0002  00?? 00052      movwf dc3
0003  0C0D 00053 dly2      movlw .13          ; repeat inner loop 13 times
0004  00?? 00054      movwf dc2          ; -> 13x(767+3)-1 = 10009 cycles
0005  00?? 00055      clrf dc1          ; inner loop = 256x3-1 = 767 cycles
0006  02?? 00056 dly1      decfsz dc1,f
0007  0A?? 00057      goto dly1
0008  02?? 00058      decfsz dc2,f          ; end middle loop
0009  0A?? 00059      goto dly1
000A  02?? 00060      decfsz dc3,f          ; end outer loop
000B  0A?? 00061      goto dly2
          00062
000C  0800 00063      retlw 0
```

The `banksel` directive is completely undefined at this point; even the instruction hasn’t been decided, so it’s shown as ‘???? ????’. It can’t be defined, because the location of ‘dc3’ is unknown.

Similarly, many of the instruction opcodes are only partially complete. The question marks can’t be filled in, until the locations of all the data and program labels are known.

Assigning locations to the various objects is the linker’s job, and you can see the choices it has made by looking at the project’s ‘.map’ *map file*. It shows where each section will be placed, and what the final data and program addresses are. For example (reformatted a little here):

Section Info					
Section	Type	Address	Location	Size (Bytes)	
RESET	code	0x000000	program	0x00000a	
.cinit	romdata	0x000005	program	0x000004	
.code	code	0x000007	program	0x00001a	
MAIN	code	0x000014	program	0x000018	
RCCAL	code	0x0003ff	program	0x000002	
.config_0FFF_BA_L3-FLASH_LED-MAIN.O	code	0x000fff	program	0x000002	
.udata_shr	udata	0x000007	data	0x000001	
.udata	udata	0x000010	data	0x000003	

Program Memory Usage	
Start	End
0x000000	0x00001f
0x0003ff	0x0003ff
0x000fff	0x000fff

⁶ With MPLAB X, you’ll find these files under folders such as “build”, within your project folder.

⁷ With MPLAB X, the linker does not, by default, generate a map file. You can change this in ‘mlink’ section of the “Project Properties” window, by specifying a file name in the ‘Generate map file’ field.

34 out of 1029 program addresses used, program memory utilization is 3%

Symbols - Sorted by Name			
Name	Address	Location	Storage File
delay10	0x000003	program	static C:\...\BA_L3-Flash_LED-main.asm
delay10_R	0x000007	program	extern C:\...\delay10.asm
dly1	0x00000d	program	static C:\...\delay10.asm
dly2	0x00000a	program	static C:\...\delay10.asm
main_loop	0x000017	program	static C:\...\BA_L3-Flash_LED-main.asm
start	0x000014	program	static C:\...\BA_L3-Flash_LED-main.asm
dc1	0x000010	data	static C:\...\delay10.asm
dc2	0x000011	data	static C:\...\delay10.asm
dc3	0x000012	data	static C:\...\delay10.asm
sGPIO	0x000007	data	static C:\...\BA_L3-Flash_LED-main.asm

These addresses are used when the linker creates the '.hex' file, containing the final assembled code, with fully resolved addresses, that will be loaded into the PIC.

Conclusion

Again, that's a lot theory, without moving far forward. We're still only flashing an LED.

The intent of this lesson was to give you an understanding of the baseline PIC memory architecture, including its limitations and how to work around them, to avoid potential problems as your programs grow. We've also seen how to create re-usable code modules, which should help you to avoid wasting time "reinventing the wheel" for each new project in future. In fact, we'll continue to use our delay module in later lessons.

In addition to providing an output (such as a blinking LED), real PIC applications usually involve responding to the environment, or at least to user input.

So, in the [next lesson](#) we'll look at reading and responding to switches, such as pushbuttons.

And since real switches "bounce", and that can be a problem for microcontroller applications, we'll look at ways to "debounce" them.

Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 4: Reading Switches

The [previous lessons](#) introduced simple digital output, by turning on or flashing an LED. That's more useful than it seems, because, with some circuit changes (such as adding transistors and relays), it can be readily adapted to turning on and off almost any electrical device.

But most systems need to interact with their environment in some way; to respond according to user commands or varying inputs. The simplest form of input is an on/off switch. This lesson shows how to read and respond to a simple pushbutton switch, or, equivalently, a slide or toggle switch, or even something more elaborate such as a mercury tilt switch, or a sensor with a digital output – anything that makes or breaks a single connection, or is “on” or “off”, “high” or “low”.

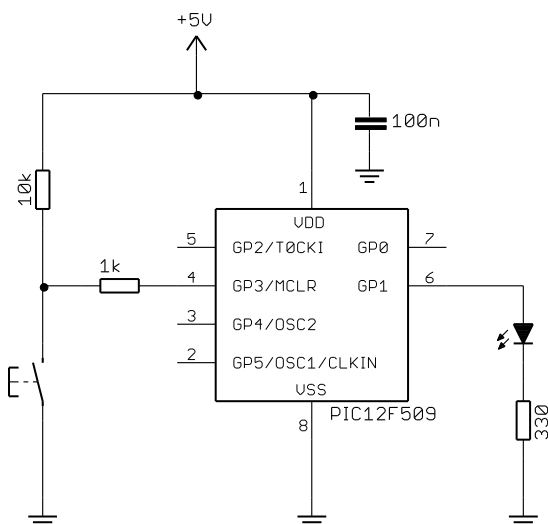
This lesson covers:

- Reading digital inputs
- Conditional branching
- Using internal pull-ups
- Hardware and software approaches to switch debouncing

Example Circuit

To show how to read a pushbutton switch, we'll need a circuit with a pushbutton!

The [Gooligum baseline training board](#) provides tact switches connected to pins GP2 and GP3, while the Microchip Low Pin Count demo board only has a tact switch connected to GP3, as in the circuit shown below, so we'll use this circuit in this lesson.



We'll continue to use a LED on GP1, as in the previous lessons. If you're using the Gooligum training board, you should connect jumper JP3, to bring the 10 kΩ resistor into the circuit.

The pushbutton is connected to GP3 via a 1 kΩ resistor. This is good practice, but not strictly necessary. Such resistors are used to provide some isolation between the PIC and the external circuit, for example to limit the impact of over- or under-voltages on the input pin, or to provide some protection against an input pin being inadvertently programmed as an output. If the switch was to be pressed while the pin was mistakenly configured as an output and set “high”, the result is

likely to be a dead PIC – unless there is a resistor to limit the current flowing to ground.

In this case, that scenario is impossible, because, as mentioned in [lesson 1](#), GP3 can only ever be an input. So why the resistor? It is necessary to allow the PIC to be safely and successfully programmed.

You might recall, from [lesson 0](#), that the PICkit 2 and PICkit 3 are In-Circuit Serial Programming (ICSP) programmers. The ICSP protocol allows the PICs that support it to be programmed while in-circuit. That is, they don't have to be removed from the circuit and placed into a separate, stand-alone programmer. That's very convenient, but it does place some restrictions on the circuit. The programmer must be able to set appropriate voltages on particular pins, without being affected by the rest of the circuit. That implies some isolation, and often a simple resistor, such as the 1 k Ω resistor here, is all that is needed.

To place a PIC such as the 12F509 into programming mode, a high voltage (around 12V) is applied to pin 4 – the same pin that is used for GP3. Imagine what would happen if, while the PIC was being programmed, with 12V applied to pin 4, that pin was grounded by someone pressing a pushbutton connected directly to it! The result in this case wouldn't be a dead PIC; it would be a dead PICkit 2 or PICkit 3 programmer... Or suppose that a sensor with a digital output was connected to the GP3 input. That sensor may not react well to having 12 V applied directly to its output!

But, if you are sure that you know what you are doing and understand the risks, you can leave out isolation or protection resistors, such as the 1 k Ω resistor on GP3.

The 10 k Ω resistor holds GP3 high while the switch is open. How can we be sure? According to the PIC12F509 data sheet, the “input leakage current” flowing into GP3 can be up to 5 μ A (parameter D061). That equates to a voltage drop of up to 55 mV across the 10 k Ω and 1 k Ω resistors in series ($5 \mu\text{A} \times 11 \text{k}\Omega$), so, with the switch open, the voltage at GP3 will be a minimum of $V_{DD} - 55 \text{ mV}$. The minimum supply voltage is 2.0 V (parameter D001), so in the worst case, the voltage at GP3 = $2.0 - 55 \text{ mV} = 1.945 \text{ V}$. The lowest input voltage guaranteed to be read as “high” is given as $0.25 V_{DD} + 0.8 \text{ V}$ (parameter D040A). For $V_{DD} = 2.0 \text{ V}$, this is $0.25 \times 2.0 \text{ V} + 0.8 \text{ V} = 1.3 \text{ V}$. That's well below the worst-case “high” input to GP3 of 1.945 V, so with these resistors, the pin is guaranteed to read as “high”, over the allowable supply voltage range.

In practice, you generally don't need to bother with such detailed analysis. As a rule of thumb, 10 k Ω is a good value for a *pull-up* resistor like this. But, it's good to know that the rule of thumb is supported by the characteristics specified in the data sheet.

When the switch is pressed, the pin is pulled to ground through the 1 k Ω resistor. Now the input leakage current flows out of GP3, giving a voltage drop across the 1 k Ω resistor of up to 5 mV ($5 \mu\text{A} \times 1 \text{k}\Omega$), so with the switch closed, the voltage at GP3 will be a maximum of 5 mV. The highest input voltage guaranteed to be read as a “low” is $0.15 V_{DD}$ (parameter D030A). For $V_{DD} = 2.0 \text{ V}$ (the worst case), this is $0.15 \times 2.0 \text{ V} = 300 \text{ mV}$. That's above the maximum “low” input to GP3 of 5 mV, so the pin is guaranteed to read as “low” when the pushbutton is pressed.

Again, that's something you come to know as a rule of thumb. With just a little experience, you'll look at a circuit like this and see immediately that GP3 is normally held high, but is pulled low if the pushbutton is pressed.

Interference from $\overline{\text{MCLR}}$

There is a potential problem with using a pushbutton on GP3; as we have seen, the same pin can instead be configured (using the PIC's configuration word) as the processor reset, $\overline{\text{MCLR}}$.

This is potentially a problem because, by default, as we saw in [lesson 1](#), MPLAB provides for control of the $\overline{\text{MCLR}}$ line through the “Release from Reset” and “Hold in Reset” menu items (MPLAB 8 only) and toolbar buttons (MPLAB 8 and MPLAB X).

That's not actually a problem if you're using a PICkit 3, because when the PICkit 3 is used as a programmer¹, its $\overline{\text{MCLR}}$ output is disconnected ("tri-stated") immediately after programming, meaning that the PICkit 3 won't affect the PIC's $\overline{\text{MCLR}}$ / GP3 input. The pull-up resistor and pushbutton are able to pull GP3 high and low, as described above.


It's different with the PICkit 2 where, by default, the PICkit 2 continues to assert control over the $\overline{\text{MCLR}}$ line and, because of the 1 k Ω isolation resistor, the 10 k Ω pull-up resistor and the pushbutton cannot overcome the PICkit 2's control of that line.

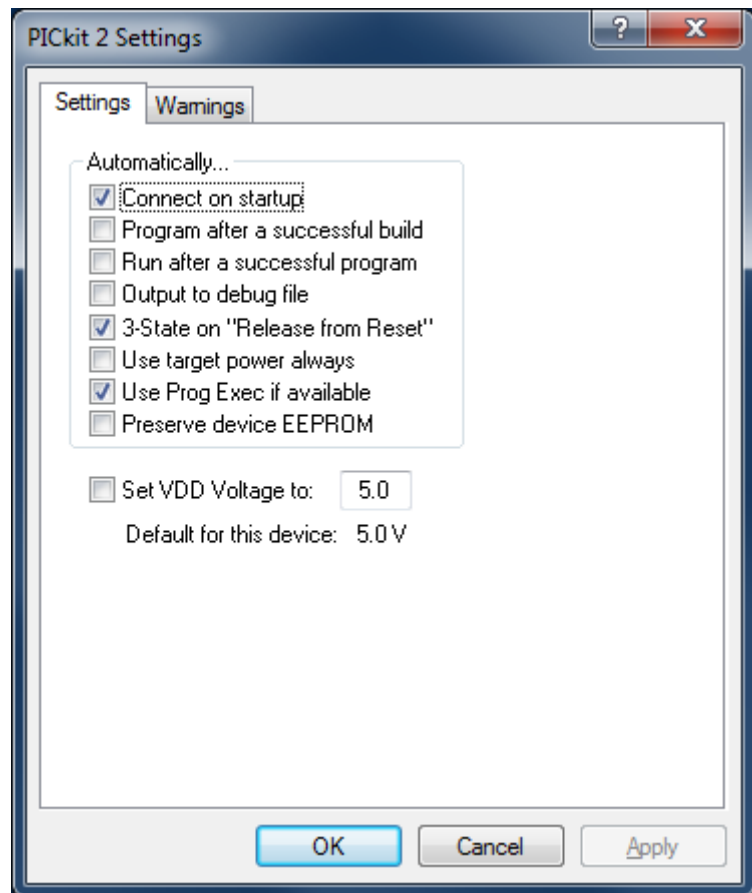
When the PICkit 2 is used as a programmer with MPLAB, it will, by default, assert control of the $\overline{\text{MCLR}}$ line, overriding any pushbutton switch on the PIC's $\overline{\text{MCLR}}$ / GP3 input.

If you are using MPLAB 8, this problem can be overcome by changing the PICkit 2 programming settings, to tri-state the PICkit 2's $\overline{\text{MCLR}}$ output (effectively disconnecting it) when it is not being used to hold the PIC in reset.

To do this, select the PICkit 2 as a programmer (using the "Programmer → Select Programmer" submenu) and then use the "Programmer → Settings" menu item to display the PICkit 2 Settings dialog window, shown on the right. Select the "3-State on "Release from Reset"" option in the Settings tab and then click "OK".

After using the PICkit 2 to program your device, it will hold $\overline{\text{MCLR}}$ low, holding the GP3 input low, overriding the pull-up resistor.

When you now click on the  button in the programming toolbar, or select the "Programmer → Release from Reset" menu item, the PICkit 2 will release control of $\overline{\text{MCLR}}$, allowing GP3 to be driven high or low by the pull-up resistor and pushbutton.

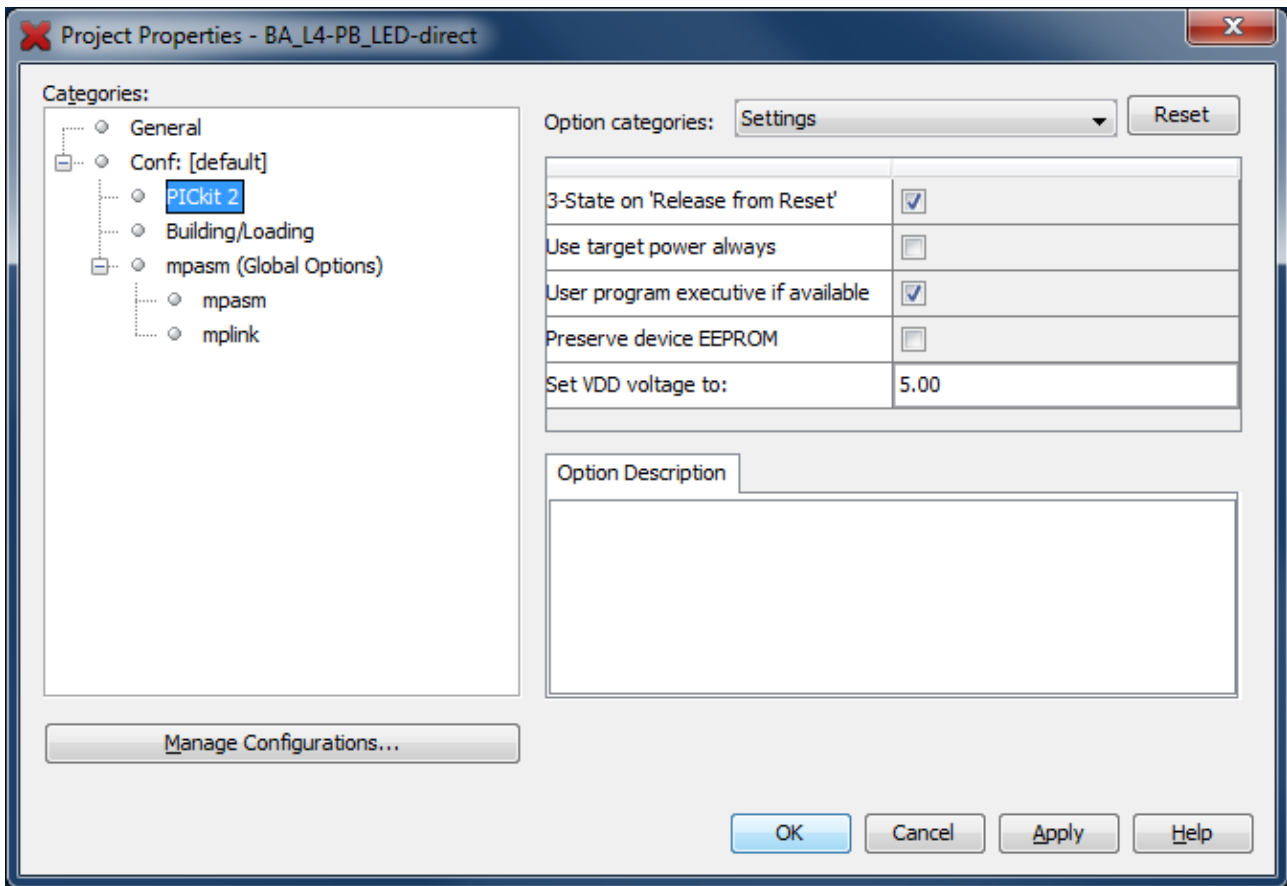


MPLAB X also allows you to prevent the PICkit 2 asserting control over $\overline{\text{MCLR}}$, in much the same way.

¹ as opposed to being used as a debugger; see [lesson 0](#)

To do this, open the Project Properties window, by selecting the “File → Project Properties” menu item, or right-clicking the project name in the Projects window and selecting “Properties”, or simply clicking on the “Project Properties” button to the left of the Dashboard.

If you click on “PICkit 2”, you will see the settings shown below:



Select “3-State on ‘Release from Reset’”, and then click “OK”.

Now, when you build and run your project, the PICkit 2’s $\overline{\text{MCLR}}$ output will be tri-stated, making it possible for you to use a pushbutton on GP3.

Reading Digital Inputs

In general, to read a pin, we need to:

- Configure the pin as an input
- Read or test the bit corresponding to the pin

Recall, from [lesson 1](#), that the pins on the PICs we’ve seen so far, including the 12F509, are all only *digital* inputs or outputs. When configured as outputs, they can be turned on or off, but nothing in between. Similarly, when configured as inputs, they can only read a voltage as being “high” or “low”.

As mentioned above, the data sheet defines input voltage ranges where the pin is guaranteed to read as “high” or “low”. For voltages between these ranges, the pin might read as either; the input behaviour for intermediate voltages is *undefined*.

As you might expect, a “high” input voltage reads as a ‘1’, and a “low” reads as a ‘0’.

To configure a pin as an input, set the corresponding TRIS bit to ‘1’.

However, as we’ve seen, GP3 is a special pin. If it is not configured as $\overline{\text{MCLR}}$, it can only be an input. It is not possible to use GP3 as an output. So, when using GP3 as an input, there’s no need to set its TRIS bit. Although for clarity, you may as well do so.

Reading an input isn’t much use unless we’re able to respond to that input – to do something different, depending on whether the input is high or low.

This is where *bit test* instructions are useful. There are two:

‘btfsc f,b’ tests bit ‘b’ in register ‘f’. If it is ‘0’, the following instruction is skipped – “bit test file register, skip if clear”.

‘btfss f,b’ tests bit ‘b’ in register ‘f’. If it is ‘1’, the following instruction is skipped – “bit test file register, skip if set”.

Their use is illustrated in the following example.

Example 1: Read a switch

We’ll start by simply lighting the LED only while the pushbutton is pressed.

Of course, that’s a waste of a microcontroller. To get the same effect, you could leave the PIC out and build the circuit shown on the right!

But, this simple example avoids having to deal with the problem of switch contact bounce, which we’ll look at later.



Here’s some code that will do this:

```
start
    movlw    b'111101'    ; configure GP1 (only) as an output
    tris     GPIO        ; (GP3 is an input)

    clrf     GPIO        ; start with GPIO clear (GP1 low)
loop
    btfss   GPIO,3      ; if button pressed (GP3 low)
    bsf     GPIO,1      ; turn on LED
    btfsc   GPIO,3      ; if button up (GP3 high)
    bcf     GPIO,1      ; turn off LED

    goto    loop        ; repeat forever
```

Note that the logic seems to be inverse; the LED is turned on if GP3 is clear, yet the `btfss` instruction tests for the GP3 bit being set. The bit test instructions skip the next instruction if the bit test condition is met, so the instruction following a bit test is executed only if the condition is *not* met. Often, following a bit test instruction, you'll place a `goto` or `call` to jump to a block of code that is to be executed if the bit test condition is not met. In this case, there is no need, as the LED can be turned on or off with single instructions:

`bsf f,b` sets bit 'b' in register 'f' to '1' – “bit set file register”.

`bcf f,b` clears bit 'b' in register 'f' to '0' – “bit clear file register”.

Previously, we have set, cleared and toggled bits by operating on the whole GPIO port at once.

That is what these bit set and clear instructions are doing behind the scenes; they read the entire port, set or clear the designated bit, and then rewrite the result. They are examples of 'read-modify-write' instructions, as discussed in [lesson 2](#), and their use can lead to unintended effects – you may find that bits other than the designated one are also being changed.

This unwanted effect often occurs when sequential bit set/clear instructions are performed on the same port. Trouble can be avoided by separating sequential `bsf` and `bcf` instructions with a `nop`.

Although unlikely to be necessary in this case, since the bit set/clear instructions are not sequential, a shadow register (see lesson 2) could be used as follows:

```
start
    movlw    b'111101'      ; configure GP1 (only) as an output
    tris    GPIO           ; (GP3 is an input)

    clrf    GPIO           ; start with GPIO clear (LED off)
    clrf    sGPIO         ; update shadow copy

loop
    btfss   GPIO,3        ; if button pressed (GP3 low)
    bsf     sGPIO,1       ; turn on LED
    btfsc   GPIO,3        ; if button up (GP3 high)
    bcf     sGPIO,1       ; turn off LED

    movf    sGPIO,w       ; copy shadow to GPIO
    movwf   GPIO

    goto    loop          ; repeat forever
```

It's possible to optimise this a little. There is no need to test for button up as well as button down; it will be either one or the other, so we can instead write a value to the shadow register, assuming the button is up, and then test just once, updating the shadow if the button is found to be down.

It's also not really necessary to initialise GPIO at the start; whatever it is initialised to, it will be updated the first time the loop completes, a few μ s later – much too fast to see.

If setting the initial values of output pins correctly is important, to avoid power-on glitches that may affect circuits connected to them, the correct values should be written to the port registers before configuring the pins as outputs, i.e. initialise GPIO before using `tris` to configure the port.

But when dealing with human perception, it's not important, so the following code is acceptable:

```
start
    movlw    b'111101'    ; configure GP1 (only) as an output
    tris    GPIO        ; (GP3 is an input)

loop
    clrf    sGPIO        ; assume button up -> LED off
    btfss   GPIO,3       ; if button pressed (GP3 low)
    bsf     sGPIO,1      ; turn on LED

    movf    sGPIO,w      ; copy shadow to GPIO
    movwf   GPIO

    goto    loop         ; repeat forever
```

If you didn't use a shadow register, but tried to take the same approach – assuming a state (such as “button up”), setting `GPIO`, then reading the button and changing `GPIO` accordingly – it would mean that the LED would be flickering on and off, albeit too fast to see. Using a shadow register is a neat solution that avoids this problem, as well as any read-modify-write concerns, since the physical register (`GPIO`) is only ever updated with the correctly determined value.

Complete program

Here's the “light an LED when button pressed” code again, along with the other parts we need to make a complete working program. Note that the comments include the statement that the pushbutton switch is “active low”, meaning that it's connected such that when it is pressed (activated), the PIC input is driven low. It makes it easier to maintain your code if you are clear about any assumptions like that.

```
*****
;
; Description: Lesson 4, example 1b
;
; Demonstrates reading a switch
; (using shadow register to update port)
;
; Turns on LED when pushbutton on GP3 is pressed
;
*****
;
; Pin assignments:
; GP1 = LED
; GP3 = pushbutton switch (active low)
;
*****

list      p=12F509
#include   <p12F509.inc>

;***** CONFIGURATION
                ; int reset, no code protect, no watchdog, int RC clock
__CONFIG      _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC

;***** VARIABLE DEFINITIONS
                UDATA_SHR
sGPIO        res 1                ; shadow copy of GPIO
```

```

;***** RC CALIBRATION
RCCAL   CODE    0x3FF           ; processor reset vector
        res 1                   ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET   CODE    0x000           ; effective reset vector
        movwf  OSCCAL           ; apply internal RC factory calibration

;***** MAIN PROGRAM *****

;***** Initialisation
start
        movlw  b'111101'       ; configure GP1 (only) as an output
        tris   GPIO            ; (GP3 is an input)

;***** Main loop
main_loop
        ; turn on LED only if button pressed
        clrf   sGPIO           ; assume button up -> LED off
        btfss  GPIO,3          ; if button pressed (GP3 low)
        bsf    sGPIO,1         ; turn on LED

        movf   sGPIO,w         ; copy shadow to GPIO
        movwf  GPIO

        ; repeat forever
        goto   main_loop

        END

```

Debouncing

In most applications, you want your code to respond to transitions; some action should be triggered when a button is pressed or a switch is toggled.

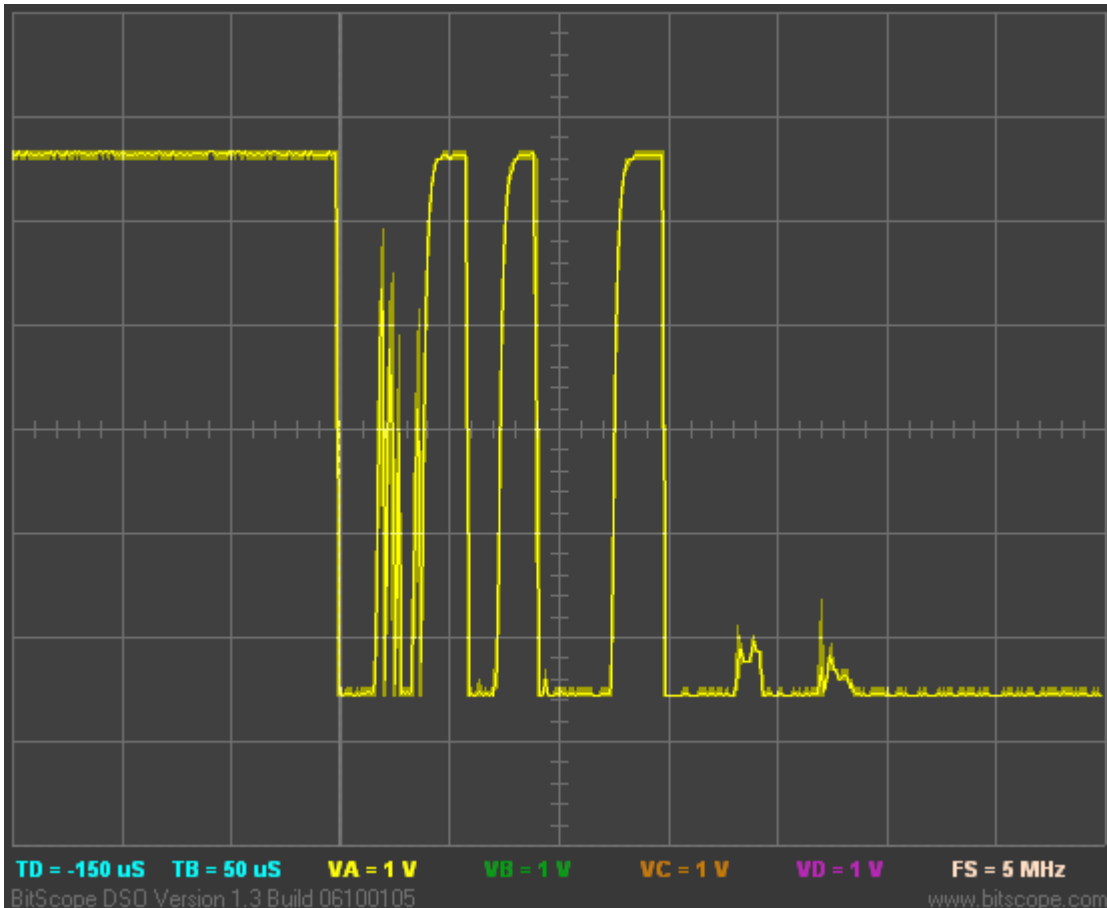
This presents a problem when interacting with real, physical switches, because their contacts *bounce*.

When most switches change, the contacts in the switch will make and break a number of times before settling into the new position. This contact bounce is generally too fast for a human to perceive, but microcontrollers are fast enough to react to each of these rapid, unwanted transitions.

Overcoming this problem is called switch *debouncing*.

There are many possible ways to address the problem of switch bounce, some of which we'll look at in this section.

The picture at the top of the next page is a recording of an actual switch bounce, using a common pushbutton switch:



The switch transitions several times before settling into the new state (low), after around 250 μ s.

A similar problem can be caused by *electromagnetic interference (EMI)*. Unwanted spikes may appear on an input line, due to electromagnetic noise, especially (but not only) when switches or sensors are some distance from the microcontroller. But any solution which deals effectively with contact bounce will generally also remove or ignore input spikes caused by EMI.

Hardware debouncing

Debouncing is effectively a filtering problem; you want to filter out fast transitions, leaving only the slower changes that result from intentional human input.

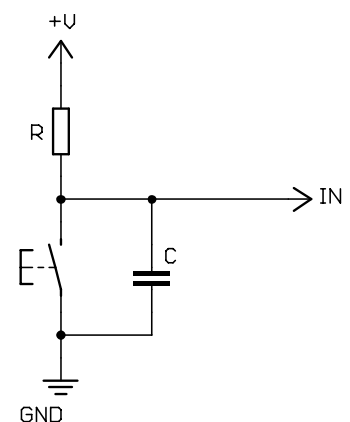
That suggests a low-pass filter; the simplest of which consists of a resistor and a capacitor (an “*RC filter*”).

To debounce a normally-open pushbutton switch, pulled high by a resistor, the simplest hardware solution is to place a capacitor directly across the switch, as shown at right.

In theory, that’s all that’s needed. The idea is that, when the switch is pressed, the capacitor is immediately discharged, and the input will go instantly to 0 V. When the contacts bounce open, the capacitor will begin to charge, through the resistor. The voltage at the input pin is the voltage across the capacitor:

$$V_{in} = V_{DD} \left(1 - e^{-t/RC} \right)$$

This is an exponential function with a *time constant* equal to the product RC.



The general I/O pins on the PIC12F509 act as TTL inputs: given a 5 V power supply, any input voltage between 0 V and 0.8 V reads as a '0'. As long as the input remains below 0.8 V, the PIC will continue to read '0', which is what we want, to avoid transitions to '1' due to switch bounce.

Solving the above equation for $V_{DD} = 5.0 \text{ V}$ and $V_{in} = 0.8 \text{ V}$ gives $t = 0.174RC$.

This is the maximum time that the capacitor can charge, before the input voltage goes higher than that allowed for a logical '0'. That is, it's the longest 'high' bounce that will be filtered out.

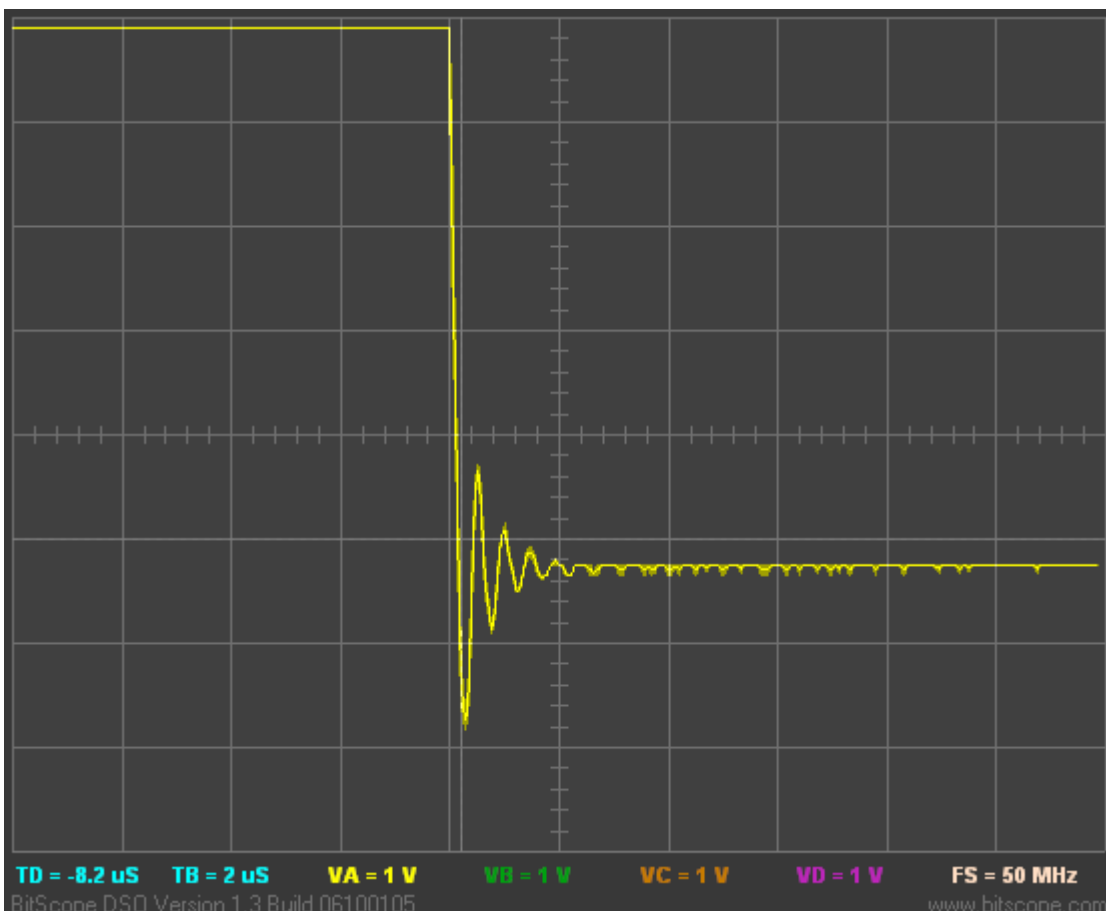
In the pushbutton press recorded above, the longest 'high' bounce is approx. $25 \mu\text{s}$. Assuming a pull-up resistance of $10 \text{ k}\Omega$, as in the original circuit, we can solve for $C = 25 \mu\text{s} \div (0.174 \times 10 \text{ k}\Omega) = 14.4 \text{ nF}$. So, in theory, any capacitor 15 nF or more could be used to effectively filter out these bounces.

In practice, you don't really need all these calculations. As a rule of thumb, if you choose a time constant several times longer than the maximum settling time ($250 \mu\text{s}$ in the switch press above), the debouncing will be effective. So, for example, 1 ms would be a reasonable time constant to aim for here – it's a few times longer than the settling time, but still well below human perception (no one will notice a 1 ms delay after pressing a button).

To create a time constant of 1 ms , you can use a $10 \text{ k}\Omega$ pull-up resistor with a 100 nF capacitor:

$$10 \text{ k}\Omega \times 100 \text{ nF} = 1 \text{ ms}$$

Testing the above circuit, with $R = 10 \text{ k}\Omega$, $C = 100 \text{ nF}$ and using the same pushbutton switch as before, gave the following response:



Sure enough, the bounces are all gone, but there is now an overshoot – a ringing at around 2 MHz , decaying in around $2 \mu\text{s}$. The problem is that the description above is idealised. In the real world, capacitors and switches and the connections between them all have resistance, so the capacitor cannot discharge instantly. More significantly, every component and interconnection has some inductance. The combination of

inductance and capacitance leads to oscillation (the 2 MHz ringing). Inductance has the effect of maintaining current flow. When the switch contacts are closed, a high current rapidly discharges the capacitor. The inductance causes this current flow to continue beyond the point where the capacitor is fully discharged, slightly charging in the opposite direction, making V_{in} go (briefly) negative. Then it reverses, the capacitor discharging in the opposite direction, overshooting again, and so on – the oscillation losing energy to resistance and quickly dying away.

So – is this a problem? Yes!

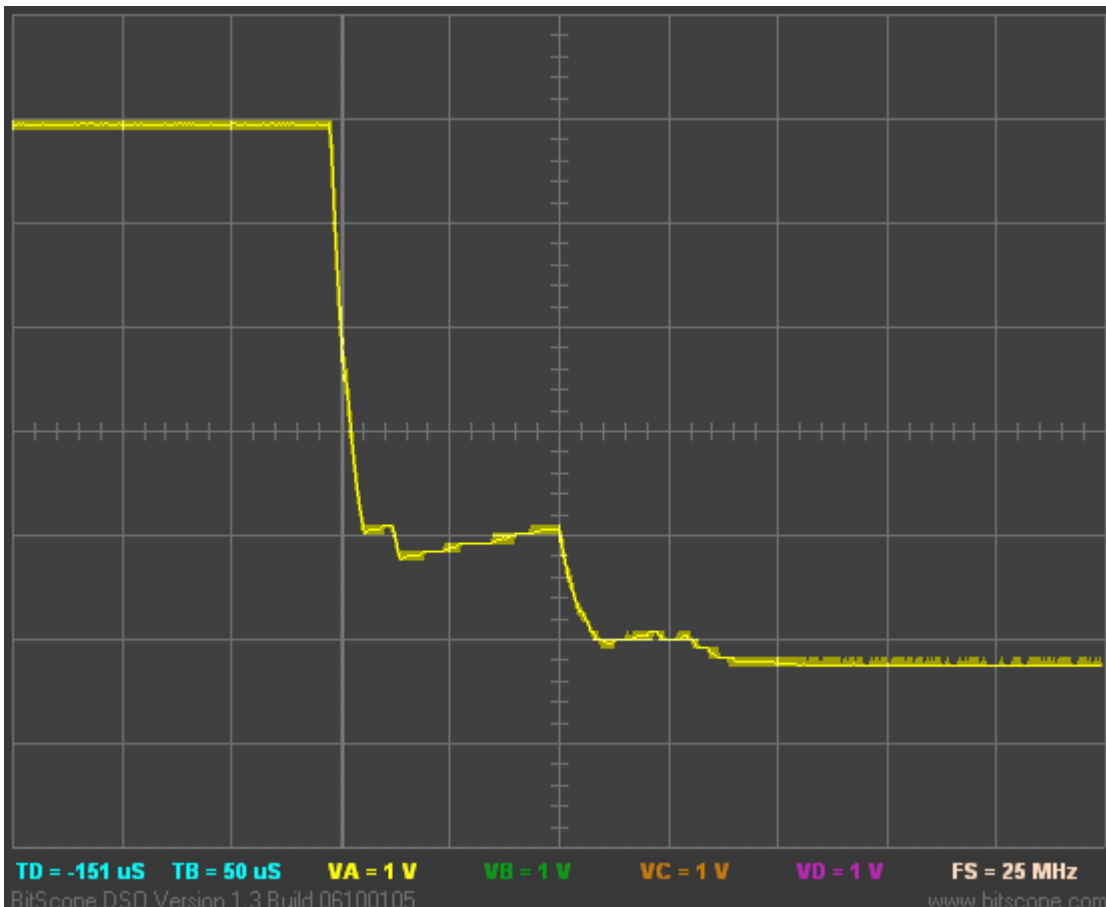
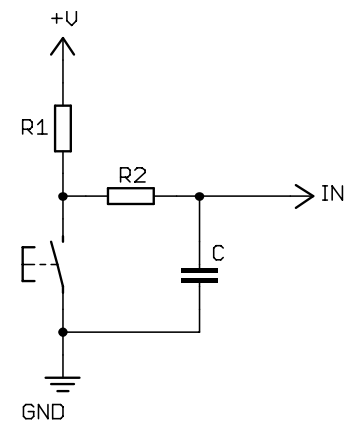
According to the PIC12F509 data sheet, the absolute minimum voltage allowed on any input pin is -0.3 V. But the initial overshoot in the pushbutton press response, shown above, is approx. -1.5 V. That means that this simple debounce circuit is presenting voltages outside the 12F509's absolute maximum ratings. You might get away with it. Or you might end up with a fried PIC!

To avoid this, we need to limit the discharge current from the capacitor, since it is the high discharge current that is working through stray inductance to drive the input voltage to a dangerously low level.

In the circuit shown at right, the discharge current is limited by the addition of resistor R2.

We still want the capacitor to discharge much more quickly than it charges, since the circuit is intended to work essentially the same way as the first – a fast discharge to 0 V, followed by much slower charging during 'high' bounces. So we should have R2 much smaller than R1.

The following oscilloscope trace shows the same pushbutton response as before, with $R1 = 10\text{ k}\Omega$, $R2 = 100\ \Omega$ and $C = 100\text{ nF}$:



The ringing has been eliminated.

Instead of large steps from low to high, the bounces show as “ramps”, of up to 75 μ s, where the voltage rises by up to 0.4 V.

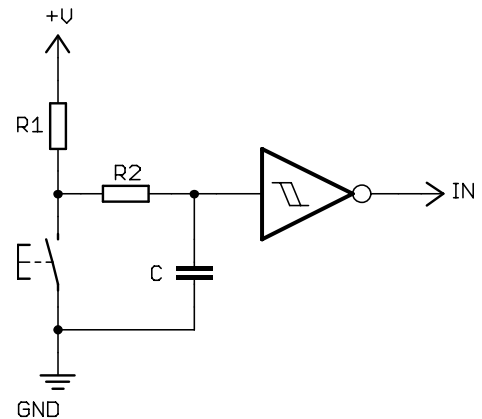
This effect could be reduced, and the decline from high to low made smoother, by adjusting the values of R1, R2 and C. But small movements up and down, of a fraction of a volt, will never be eliminated. And the fact that the high \rightarrow low transition takes time to settle can present a problem.

With a 5 V power supply, according to the PIC12F509 data sheet, a voltage between 0.8 V and 2.0 V on a TTL-compatible input (any of the general I/O pins) is undefined. Voltages between 0.8 V and 2.0 V could read as either a ‘0’ or a ‘1’. If we can’t guarantee what value will be read, we can’t say that the switch has been debounced; it’s still an unknown.

An effective solution to this problem is to use a Schmitt trigger buffer, such as a 74HC14 inverter, as shown in the circuit on the right.

A Schmitt trigger input displays *hysteresis*; on the high \rightarrow low transition, the buffer will not change state until the input falls to a low voltage threshold (say 0.8 V). It will not change state until the input rises to a high voltage threshold (say 1.8 V).

That means that, given a slowly changing input signal, which is generally falling, with some small rises on the way down (as in the trace above), only a single transition will appear at the buffer’s output. Similarly, a Schmitt trigger will clean up slowly rising, noisy input signal, producing a single sharp transition, at the correct TTL levels, suitable for interfacing directly to the PIC.



Of course, if you use a Schmitt trigger inverter, as shown here, you must reverse your program’s logic: when the switch is pressed, the PIC will see a ‘1’ instead of a ‘0’.

Note that when some of the PIC12F509’s pins are configured for special function inputs, instead of general purpose inputs, they use Schmitt trigger inputs. For example, as we’ve seen, pin 4 of the 12F509 can be configured as an external reset line ($\overline{\text{MCLR}}$) instead of GP3. When connecting a switch for external $\overline{\text{MCLR}}$ you only need an RC filter for debouncing; the Schmitt trigger input is built into the reset circuitry on the PIC.

Software debouncing

One of the reasons to use microcontrollers is that they allow you to solve what would otherwise be a hardware problem, in software. A good example is switch debouncing.

If the software can ignore input transitions due to contact bounce or EMI, while responding to genuine switch changes, no external debounce circuitry is needed. As with the hardware approach, the problem is essentially one of filtering; we need to ignore any transitions too short to be ‘real’.

But to illustrate the problem, and provide a base to build on, we’ll start with some code with no debouncing at all.

Suppose the task is to toggle the LED on GP1, once, each time the button on GP3 is pressed.

In pseudo-code, this could be expressed as:

```
do forever
    wait for button press
    toggle LED
    wait for button release
end
```

Note that it is necessary to wait for the button to be released before restarting the loop. This ensures that the LED will toggle only once per button press. If we didn't wait for the button to be released before continuing, the LED would continue to toggle as long as the button was held down; not the desired behaviour.

This pseudo-code could be implemented as:

```

loop
    ; wait for button press
wait_dn btfsc    GPIO,3          ; wait until GP3 low
        goto    wait_dn

        ; toggle LED on GP1
        movf    sGPIO,w
        xorlw   b'000010'       ; toggle bit corresponding to GP1 (bit 1)
        movwf   sGPIO           ; in shadow register
        movwf   GPIO            ; and write to GPIO

        ; wait for button release
wait_up btfss    GPIO,3          ; wait until GP3 high
        goto    wait_up

        ; repeat forever
        goto    loop

```

If you build this into a complete program² and test it, you will find that it is difficult to reliably change the LED when you press the button; sometimes it will change, other times not. This is due to contact bounce.

Debounce delay

A simple approach to software is to estimate the maximum time the switch could possibly take to settle, and then merely wait at least that long, after detecting the first transition. If the wait time, or delay, is longer than the maximum possible settling time, you can be sure that after this delay the switch will have finished bouncing.

It's only a matter of adding a suitable debounce delay, after each transition is detected, as in the following pseudo-code:

```

do forever
    wait for button press
    toggle LED
    delay debounce_time
    wait for button release
    delay debounce_time
end

```

Note that the LED is toggled immediately after the button press is detected. There's no need to wait for debouncing. By acting on the button press as soon as it is detected, the user will experience as fast a response as possible.

But it is important to ensure that the "button pressed" state is stable (debounced), before waiting for button release. Otherwise, the first bounce after the button press would be seen as a release.

The necessary minimum delay time depends on the characteristics of the switch. For example, the switch tested above was seen to settle in around 250 μ s. Repeated testing showed no settling time greater than 1 ms, but it's difficult to be sure of that, and perhaps a different switch, say that used in production hardware, rather than the prototype, may behave differently. So it's best to err on the safe side, and use the longest

² You'd need to add processor configuration, reset vector, initialisation code etc., and declare the `sGPIO` variable, as we've done before. Or download the complete source code from www.gooligum.com.au.

delay we can get away with. People don't notice delays of 20 ms or less (flicker is only barely perceptible at 50 Hz, corresponding to a 20 ms delay), so a good choice is probably 20 ms.

As you can see, choosing a suitable debounce delay is not an exact science!

The previous code can be modified to call the 10 ms delay module we developed in [lesson 3](#), as follows:

```
main_loop
    ; wait for button press
wait_dn btfsc  GPIO,3          ; wait until GP3 low
        goto   wait_dn

        ; toggle LED on GP1
        movf   sGPIO,w
        xorlw  b'000010'      ; toggle bit corresponding to GP1 (bit 1)
        movwf  sGPIO          ; in shadow register
        movwf  GPIO           ; and write to GPIO

        ; delay to debounce button press
        movlw  .2
        pagesel delay10
        call   delay10        ; delay 2 x 10 ms = 20 ms
        pagesel $

        ; wait for button release
wait_up btfss  GPIO,3          ; wait until GP3 high
        goto   wait_up

        ; delay to debounce button press
        movlw  .2
        pagesel delay10
        call   delay10        ; delay 2 x 10 ms = 20 ms
        pagesel $

        ; repeat forever
        goto   main_loop
```

If you build and test this code, you should find that the LED now reliably changes state every time you press the button.

Counting algorithm

There are a couple of problems with using a fixed length delay for debouncing.

Firstly, the need to be “better safe than sorry” means making the delay as long as possible, and probably slowing the response to switch changes more than is really necessary, potentially affecting the feel of the device you're designing.

More importantly, the delay approach cannot differentiate between a glitch and the start of a switch change. As discussed, spurious transitions can be caused by EMI, or electrical noise – or a momentary change in pressure while a button is held down.

A commonly used approach, which avoids these problems, is to regularly read (or *sample*) the input, and only accept that the switch is in a new state, when the input has remained in that state for some number of times in a row. If the new state isn't maintained for enough consecutive times, it's considered to be a glitch or a bounce, and is ignored.

For example, you could sample the input every 1 ms, and only accept a new state if it is seen 10 times in a row; i.e. high or low for a continuous 10 ms.

To do this, set a counter to zero when the first transition is seen. Then, for each sample period (say every 1 ms), check to see if the input is still in the desired state and, if it is, increment the counter before checking

again. If the input has changed state, that means the switch is still bouncing (or there was a glitch), so the counter is set back to zero and the process restarts. The process finishes when the final count is reached, indicating that the switch has settled into the new state.

The algorithm can be expressed in pseudo-code as:

```
count = 0
while count < max_samples
    delay sample_time
    if input = required_state
        count = count + 1
    else
        count = 0
end
```

Here is the modified “toggle an LED” loop, illustrating the use of this counting debounce algorithm:

```
main_loop
    banksel db_cnt          ; select data bank for this section

    ; wait for button press
db_dn  clrfsz db_cnt        ; wait until button pressed (GP3 low)
       clrfsz dcl          ; debounce by counting:
dn_dly incfsz dcl,f        ; delay 256x3 = 768 us.
       goto dn_dly
       btfsc GPIO,3        ; if button up (GP3 high),
       goto db_dn          ; restart count
       incf db_cnt,f        ; else increment count
       movlw .13           ; max count = 10ms/768us = 13
       xorwf db_cnt,w       ; repeat until max count reached
       btfss STATUS,Z
       goto dn_dly

    ; toggle LED on GP1
       movf sGPIO,w
       xorlw b'000010'     ; toggle bit corresponding to GP1 (bit 1)
       movwf sGPIO         ; in shadow register
       movwf GPIO          ; and write to GPIO

    ; wait for button release
db_up  clrfsz db_cnt        ; wait until button released (GP3 high)
       clrfsz dcl          ; debounce by counting:
up_dly incfsz dcl,f        ; delay 256x3 = 768 us.
       goto up_dly
       btfss GPIO,3        ; if button down (GP3 low),
       goto db_up          ; restart count
       incf db_cnt,f        ; else increment count
       movlw .13           ; max count = 10ms/768us = 13
       xorwf db_cnt,w       ; repeat until max count reached
       btfss STATUS,Z
       goto up_dly

    ; repeat forever
       goto main_loop
```

There are two debounce routines here; one for the button press, the other for button release. The program first waits for a pushbutton press, debounces the press, and then toggles the LED before waiting for the pushbutton to be released, and then debouncing the release.

The only difference between the two debounce routines is the input test: ‘btfsc GPIO, 3’ when testing for button up, versus ‘btfss GPIO, 3’ to test for button down.

The above code demonstrates one method for counting up to a given value (13 in this case):

The count is zeroed at the start of the routine.

It is incremented within the loop, using the ‘`incf`’ instruction – “**increment file register**”. As with many other instructions, the incremented result can be written back to the register, by specifying ‘`f`’ as the destination, or to *W*, by specifying ‘`w`’ – but normally you would use it as shown, with ‘`f`’, so that the count in the register is incremented.

The baseline PICs also provide a ‘`decf`’ instruction – “**decrement file register**”, which is similar to ‘`incf`’, except that it performs a decrement instead of increment.

We’ve seen the ‘`xorwf`’ instruction before, but not used in quite this way. The result of exclusive-oring any binary number with itself is zero. If any dissimilar binary numbers are exclusive-ored, the result will be non-zero. Thus, XOR can be used to test for equality, which is how it is being used here. First, the maximum count value is loaded into *W*, and then this maximum count value in *W* is xor’d with the loop count. If the loop counter has reached the maximum value, the result of the XOR will be zero. Note that we do not care what the result of the XOR actually is, only whether it is zero or not. And we certainly do not want to overwrite the loop counter with the result, so we specify ‘`w`’ as the destination of the ‘`xorwf`’ instruction – writing the result to *W*, effectively discarding it.

To check whether the result of the XOR was zero (which will be true if the count has reached the maximum value), we use the ‘`btfsz`’ instruction to test the zero flag bit, *Z*, in the **STATUS** register.

Alternatively, the debounce loop could have been coded by initialising the loop counter to the maximum value at the start of the loop, and using ‘`decfsz`’ at the end of the loop, as follows:

```

; wait for button press, debounce by counting:
db_dn  movlw   .13           ; max count = 10ms/768us = 13
        movwf  db_cnt
        clrf   dcl
dn_dly  incfsz  dcl,f         ; delay 256x3 = 768 us.
        goto  dn_dly
        btfsz  GPIO,3       ; if button up (GP3 high),
        goto  db_dn         ; restart count
        decfsz db_cnt,f     ; else repeat until max count reached
        goto  dn_dly

```

That’s two instructions shorter, and at least as clear, so it’s a better way to code this routine.

But in some situations it is better to count up to a given value, so it’s also worth knowing how to do that, including the use of XOR to test for equality, as shown above.

Complete program

Substituting this new debounce routine into the previous “toggle an LED” loop, and wrapping it in the other pieces we need to create a full working program, we get:

```

;*****
;
; Description: Lesson 4, example 2d
;
; Demonstrates use of counting algorithm for debouncing
;
; Toggles LED when pushbutton is pressed then released,
; using a counting algorithm to debounce switch
; (alternative version using decfsz in debounce loop)
;
;*****
;
; Pin assignments:

```

```

;      GP1 = LED                                     *
;      GP3 = pushbutton switch (active low)         *
;                                                    *
;*****
list      p=12F509
#include   <p12F509.inc>

;***** CONFIGURATION
                ; int reset, no code protect, no watchdog, int RC clock
__CONFIG    _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntrRC_OSC

;***** VARIABLE DEFINITIONS
                UDATA_SHR
sGPIO      res 1                ; shadow copy of GPIO

                UDATA
db_cnt     res 1                ; debounce counter
dcl        res 1                ; delay counter

;***** RC CALIBRATION
RCCAL      CODE    0x3FF        ; processor reset vector
                res 1                ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET      CODE    0x000        ; effective reset vector
                movwf   OSCCAL      ; apply internal RC factory calibration

;***** MAIN PROGRAM *****

;***** Initialisation
start
                clrf    GPIO        ; start with LED off
                clrf    sGPIO       ; update shadow
                movlw   b'111101'   ; configure GP1 (only) as an output
                tris    GPIO        ; (GP3 is an input)

;***** Main loop
main_loop
                banksel db_cnt      ; select data bank for this section

                ; wait for button press, debounce by counting:
db_dn      movlw   .13             ; max count = 10ms/768us = 13
                movwf  db_cnt
                clrf   dcl
dn_dly     incfsz  dcl,f           ; delay 256x3 = 768 us.
                goto   dn_dly
                btfsc  GPIO,3      ; if button up (GP3 high),
                goto   db_dn        ; restart count
                decfsz db_cnt,f     ; else repeat until max count reached
                goto   dn_dly

                ; toggle LED on GP1
                movf   sGPIO,w
                xorlw  b'000010'   ; toggle bit corresponding to GP1 (bit 1)
                movwf  sGPIO       ; in shadow register
                movwf  GPIO        ; and write to GPIO

```

```

; wait for button release, debounce by counting:
db_up   movlw   .13           ; max count = 10ms/768us = 13
        movwf  db_cnt
        clrf   dcl
up_dly  incfsz  dcl,f         ; delay 256x3 = 768 us.
        goto  up_dly
        btfss GPIO,3        ; if button down (GP3 low),
        goto  db_up         ; restart count
        decfsz db_cnt,f     ; else repeat until max count reached
        goto  up_dly

; repeat forever
        goto  main_loop

```

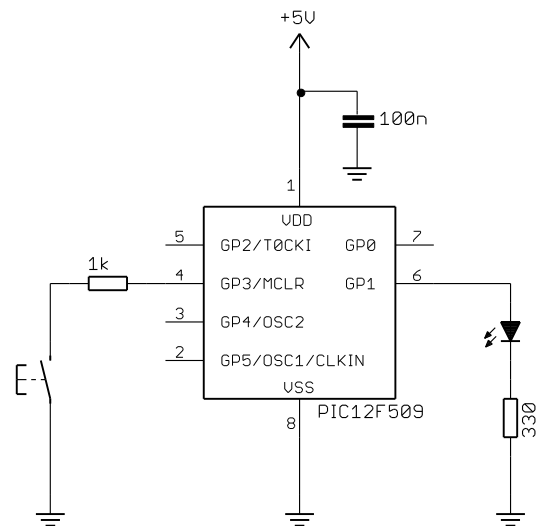
END

Internal Pull-ups

The use of pull-up resistors is so common that most modern PICs make them available internally, on at least some of the pins.

By using internal pull-ups, we can do away with the external pull-up resistor, as shown in the circuit on the right.

Strictly speaking, the internal pull-ups are not simple resistors. Microchip refer to them as “weak pull-ups”; they provide a small current which is enough to hold a disconnected, or *floating*, input high, but does not strongly resist any external signal trying to drive the input low. This current is typically 250 μA on most input pins (parameter D070), or up to 30 μA on GP3, when configured with internal pull-ups enabled.



The internal weak pull-ups are controlled by the $\overline{\text{GPPU}}$ bit in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION	$\overline{\text{GPWU}}$	$\overline{\text{GPPU}}$	T0CS	T0SE	PSA	PS2	PS1	PS0

The OPTION register is used to control various aspects of the PIC’s operation, including the timer (which will be introduced in the [next lesson](#)), as well as weak pull-ups.

Like TRIS, OPTION is not directly addressable, is write-only, and can only be written using a special instruction: ‘option’ – “load **option** register”.

By default (after a power-on or reset), $\overline{\text{GPPU}} = 1$ and the internal pull-ups are disabled. To enable internal pull-ups, clear $\overline{\text{GPPU}}$.

Assuming no other options are being set (leaving all the other bits at the default value of ‘1’), internal pull-ups are enabled by:

```

movlw   b'10111111'      ; enable internal pull-ups
        ; -0-----      ; pullups enabled (/GPPU = 0)
option

```

Note the way that this has been commented: the line with ‘-0-----’ makes it clear that only bit 6 ($\overline{\text{GPPU}}$) is relevant to enabling or disabling the internal pull-ups, and that they are enabled by clearing this bit.

The initialisation code from the last example now becomes:

```

;***** Initialisation
start
    movlw    b'10111111'    ; enable internal pull-ups
                    ; -0-----    pullups enabled (/GPPU = 0)
    option
    clrf    GPIO            ; start with LED off
    clrf    sGPIO          ; update shadow
    movlw    b'111101'     ; configure GP1 (only) as an output
    tris    GPIO           ; (GP3 is an input)

```

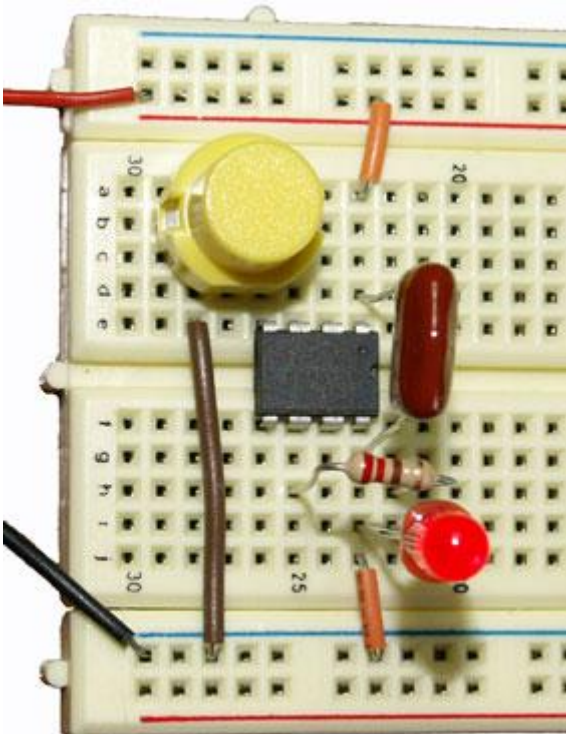
The rest of the program remains the same as before – the only difference is that the internal weak pull-ups have been enabled.

In the PIC12F509, internal pull-ups are only available on GP0, GP1 and GP3.

Internal pull-ups on the baseline PICs are not selectable by pin; they are either all on, or all off. However, if a pin is configured as an output, the internal pull-up is disabled for that pin (preventing excess current being drawn).

To test that the internal pull-ups are working, you will need to remove the 10 kΩ external pull-up resistor from the circuit we used previously.

If you have the Gooligum baseline training board, simply remove jumper JP3 and the pull-up resistor is disconnected from the pushbutton on GP3.



If you're using the Microchip Low Pin Count Demo Board, there's no easy way to disconnect the pull-up resistor from the pushbutton on that board.

One option is to build the above circuit (without a pull-up resistor), using prototyping breadboard, as illustrated on the left.

You would use the LPC demo board to program the 12F509, as usual, but then, when the PIC is programmed, remove it from the demo board and place into your breadboarded circuit.

Note that, unlike the circuit above, the prototype illustrated here does not include a current-limiting resistor between GP3 and the pushbutton. As discussed earlier, that's generally ok, but to be safe, it's good practice to include a current limiting resistor, of around 1 kΩ, between the PIC pin and the pushbutton.

But as this example illustrates, functional PIC-based circuits really can need very few external components!

When you have a version of the circuit without an external pull-up resistor, you should try testing it with the program from the previous example. You will find that the program no longer responds to the pushbutton.

However, if you modify the initialisation code, adding the instructions to enable the weak pull-ups, you will find that the program now responds correctly again – even with no external pull-up resistor!

Conclusion

You should now be able to write programs which read and respond to simple switches or other digital inputs, and be able to effectively debounce switch or other noisy inputs.

As an exercise, you could try modifying the examples in this lesson, to use a different pin as an input, instead of GP3. Hint: change the bit in GPIO being tested by the bit test instructions.

If you have the Gooligum baseline training board, you already have a pushbutton switch on GP2, making it easy to use as an input: jumper JP7 is used to connect a pull-up resistor to this switch. Note however that, because there is no internal weak pull-up available for GP2, you can't use GP2 for the final example. To test weak pull-ups, without using GP3, you would need to add a pushbutton switch to the training board's breadboard area, and connect it to GP0 (the only other pin with weak pull-ups is GP1, but you're already using that to drive the LED...)³.

That's it for reading switches for now. There's plenty more to explore, of course, such as reading keypads and debouncing multiple switch inputs – topics to explore later.

But in the [next lesson](#) we'll look at the PIC12F509's timer module.

³ If you use a switch with a weak pull-up on GP0 or GP1, you will have to unplug the PICkit 2 or PICkit 3 from the training board, and power the board externally while testing, because the PICkit 2 or 3 puts too much load on those pins, more than the internal weak pull-ups can overcome.

Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 5: Using Timer0

The lessons until now have covered the essentials of baseline PIC microcontroller operation: controlling digital outputs, timed via programmed delays, with program flow responding to digital inputs. That's all you really need to perform a great many tasks; such is the versatility of these devices. But PICs (and most other microcontrollers) offer a number of additional features that make many tasks much easier. Possibly the most useful of all are *timers*; so useful that at least one is included in every current 8-bit PIC.

A timer is simply a counter, which increments automatically. It can be driven by the processor's instruction clock, in which case it is referred to as a *timer*, incrementing at some predefined, steady rate. Or it can be driven by an external signal, where it acts as a *counter*, counting transitions on an input pin. Either way, the timer continues to count, independently, while the PIC performs other tasks.

And that is why timers are so very useful. Most programs need to perform a number of concurrent tasks; even something as simple as monitoring a switch while flashing an LED. The execution path taken within a program will generally depend on real-world inputs. So it is very difficult in practice to use programmed delay loops, as in [lesson 2](#), as an accurate way to measure elapsed time. But a timer will just keep counting, steadily, while your program responds to various inputs, performs calculations, or whatever.

As we'll see when we look at mid-range PICs, timers are commonly used to drive *interrupts* (routines which interrupt the normal program flow) to allow regularly timed "background" tasks to run. The baseline architecture doesn't support interrupts, but, as we'll see, timers are nevertheless very useful.

This lesson covers:

- Introduction to the Timer0 module
- Creating delays with Timer0
- Debouncing via Timer0
- Using Timer0 counter mode with an external clock

Timer0 Module

The baseline PICs provide only a single timer, referred to these days as Timer0. It used to be called the Real Time Clock Counter (RTCC), and you will find it called RTCC in some older literature. When Microchip released more advanced PICs, with more than one timer, they started to refer to the RTCC as Timer0.

Timer0 is very simple. The visible part is a single 8-bit register, TMR0, which holds the current value of the timer. It is readable and writeable. If you write a value to it, the timer is reset to that value and then starts incrementing from there. When it has reached 255, it rolls over to 0, and then continues to increment.

In the baseline architecture, there is no "overflow flag" to indicate that TMR0 has rolled over from 255 to 0; the only way to check the status of the timer is to read TMR0.

As mentioned above, TMR0 can be driven by either the instruction clock (FOSC/4) or an external signal.

The configuration of Timer0 is set by a number of bits in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION	GPWU	GPPU	T0CS	T0SE	PSA	PS2	PS1	PS0

The clock source is selected by the T0CS bit:

T0CS = 0 selects timer mode, where TMR0 is incremented at a fixed rate by the instruction clock.

T0CS = 1 selects counter mode, where TMR0 is incremented by an external signal, on the T0CKI pin. On the PIC12F508/9, this is physically the same pin as GP2.

Note that if T0CS is set to '1', it overrides the TRIS setting for GP2. That is, GP2 cannot be used as an output until T0CS is cleared. All the OPTION bits are set to '1' at power on, so you must remember to clear T0CS before using GP2 as an output. Instances like this, where multiple functions are mapped to a single pin, can be a trap for beginners, so be careful! These "traps" are often highlighted in the data sheets, so read them carefully!

T0CKI is a Schmitt Trigger input, meaning that it can be driven by and will respond cleanly to a smoothly varying input voltage (e.g. a sine wave), even with a low level of superimposed noise; it doesn't have to be a sharply defined TTL-level signal, as required by the GP inputs.

In counter mode, the T0SE bit selects whether Timer0 responds to rising or falling signals ("edges") on T0CKI. Clearing T0SE to '0' selects the rising edge; setting T0SE to '1' selects the falling edge.

Prescaler

By default, the timer increments by one for every instruction cycle (in timer mode) or transition on T0CKI (in counter mode). If timer mode is selected, and the processor is clocked at 4 MHz, the timer will increment at the instruction cycle rate of 1 MHz. That is, TMR0 will increment every 1 μ s. Thus, with a 4 MHz clock, the maximum period that Timer0 can measure directly, by default, is 255 μ s.

To measure longer periods, we need to use the *prescaler*.

The prescaler sits between the clock source and the timer. It is used to reduce the clock rate seen by the timer, by dividing it by a power of two: 2, 4, 8, 16, 32, 64, 128 or 256.

To use the prescaler with Timer0, clear the PSA bit to '0'.

[If PSA = 1, the prescaler is instead assigned to the watchdog timer – a topic covered in [lesson 7](#).]

When assigned to Timer0, the prescale ratio is set by the PS<2:0> bits, as shown in the following table:

PS<2:0> bit value	Timer0 prescale ratio
000	1 : 2
001	1 : 4
010	1 : 8
011	1 : 16
100	1 : 32
101	1 : 64
110	1 : 128
111	1 : 256

If PSA = 0 (assigning the prescaler to Timer0) and PS<2:0> = '111' (selecting a ratio of 1:256), TMR0 will increment every 256 instruction cycles in timer mode. Given a 1 MHz instruction cycle rate, the timer would increment every 256 μ s.

Thus, when using the prescaler with a 4 MHz processor clock, the maximum period that Timer0 can measure directly is $255 \times 256 \mu\text{s} = 65.28\text{ms}$.

Note that the prescaler can also be used in counter mode, in which case it divides the external signal on T0CKI by the prescale ratio.

If you don't want to use the prescaler with Timer0, set PSA to '1'.

To make all this theory clearer (hopefully!), here are some practical examples...

Timer Mode

The examples in this section demonstrate the use of Timer0 in timer mode, to:

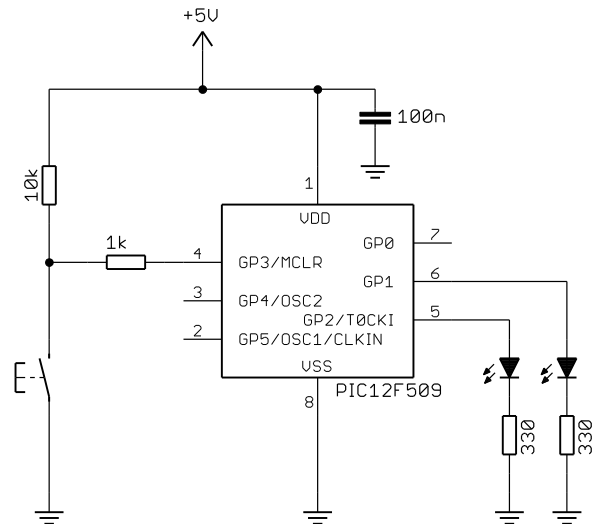
- Measure elapsed time
- Perform a regular task while responding to user input
- Debounce a switch

For each of these, we'll use the circuit shown on the right, which adds an LED to the circuit used in [lesson 4](#).

The second LED has been added to GP2, although any of the unused pins would have been suitable.

If you have the [Gooligum baseline training board](#), connect jumpers JP3, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP1 and GP2.

If you are using Microchip's Low Pin Count Demo Board, you will need to connect LEDs to GP1 and GP2, as described in [lesson 1](#).



Example 1: Reaction Timer

To illustrate how Timer0 can be used to measure elapsed time, we'll implement a very simple reaction time "game": wait a couple of seconds then light an LED to indicate 'start'. If the button is pressed within a predefined time (say 200 ms) light the other LED to indicate 'success'. If the user is too slow, leave the 'success' LED unlit. Either way, delay another second before turning off the LEDs and restarting.

There are many enhancements we could add, to make this a better game. For example, success/fail could be indicated by a bi-colour red/green LED. The delay prior to the 'start' indication should be random, so that it's difficult to cheat by predicting when it's going to turn on. The difficulty level could be made adjustable, and the measured reaction time in milliseconds could be displayed, using 7-segment displays. You can probably think of more – but the intent of here is to keep it as simple as possible, while providing a real-world example of using Timer0 to measure elapsed time.

We'll use the LED on GP2 as the 'start' signal and the LED on GP1 to indicate "success".

The program flow can be illustrated in pseudo-code as:

```
do forever
    turn off both LEDs
    delay 2 sec
    indicate start
    clear timer
    wait up to 1 sec for button press
    if button pressed and elapsed time < 200ms
        indicate success
    delay 1 sec
end
```

A problem is immediately apparent: even with maximum prescaling, Timer0 can only measure up to 65 ms. To overcome this, we need to extend the range of the timer by adding a counter variable, which is incremented when the timer overflows. That means monitoring the value in TMR0 and incrementing the counter variable when TMR0 reaches a certain value.

This example utilises the (nominally) 4 MHz internal RC clock, giving an instruction cycle time of (approximately) 1 μ s. Using the prescaler, with a ratio of 1:32, means that the timer increments every 32 μ s. If we clear TMR0 and then wait until TMR0 = 250, 8 ms ($250 \times 32 \mu$ s) will have elapsed. If we then reset TMR0 and increment a counter variable, we've implemented a counter which increments every 8 ms. Since $25 \times 8 \text{ ms} = 200 \text{ ms}$, 200 ms will have elapsed when the counter reaches 25. Hence, any counter value > 25 means the allowed time has been exceeded. And since $125 \times 8 \text{ ms} = 1 \text{ s}$, one second will have elapsed when the counter reaches 125, and we can stop waiting for the button press.

The following code sets Timer0 to timer mode (freeing GP2 to be used as an output), with the prescaler assigned to Timer0, with a 1:32 prescale ratio by:

```
movlw    b'11010100'    ; configure Timer0:
          ; --0-----    timer mode (T0CS = 0)
          ; ----0----    prescaler assigned to Timer0 (PSA = 0)
          ; -----100    prescale = 32 (PS = 100)
option   ; -> increment every 32 us
```

Assuming a 4 MHz clock, such as the internal RC oscillator, TMR0 will increment every 32 μ s.

To generate an 8 ms delay, we can clear TMR0 and then wait until it reaches 250, as follows:

```
          clrfs          TMR0          ; clear Timer0
w_tmr0   movf           TMR0,w         ; wait for 8 ms
          xorlw         .250          ; (250 ticks x 32 us/tick = 8 ms)
          btfss        STATUS,Z
          goto         w_tmr0
```

Note that XOR is used to test for equality (TMR0 = 250), as we did in [lesson 4](#).

In itself, that's an elegant way to create a delay; it's much shorter and simpler than "busy loops", such as the delay routines from lessons [2](#) and [3](#).

But the real advantage of using a timer is that it keeps ticking over, at the same rate, while other instructions are executed. That means that additional instructions can be inserted into this "timer wait" loop, without affecting the timing – within reason; if this extra code takes too long to run, the timer may increment more than once before it is checked at the end of the loop, and the loop may not finish when intended.

With 32 instruction cycles per timer increment, it's safe to insert a short piece of code to check whether the pushbutton has been checked, without risk of skipping a timer increment.

For example:

```
          clrfs          TMR0          ; clear Timer0
w_tmr0   ; repeat for 8 ms:
          btfss        GPIO,3        ; if button pressed (GP3 low)
          goto         wait_end       ; finish delay loop immediately
          movf         TMR0,w         ;
          xorlw         .250          ; (250 ticks x 32 us/tick = 8 ms)
          btfss        STATUS,Z
          goto         w_tmr0
wait_end
```

This timer loop code can then be embedded into an outer loop which increments a variable used to count the number of 8 ms periods, as follows:

```

        banksel cnt_8ms          ; clear timer (8 ms counter)
        clrf   cnt_8ms          ; repeat for 1 sec:
wait1s  clrf   TMR0            ; clear Timer0
w_tmr0  ; repeat for 8 ms:
        btfss GPIO,3           ; if button pressed (GP3 low)
        goto  wait1s_end       ; finish delay loop immediately
        movf   TMR0,w
        xorlw  .250            ; (250 ticks x 32 us/tick = 8 ms)
        btfss STATUS,Z
        goto  w_tmr0
        incf   cnt_8ms,f       ; increment 8 ms counter
        movlw .125            ; (125 x 8 ms = 1 sec)
        xorwf  cnt_8ms,w
        btfss STATUS,Z
        goto  wait1s
wait1s_end

```

The test at the end of the outer loop ($\text{cnt_8ms} = 125$) ensures that the loop completes when one second has elapsed, if the button has not yet been pressed.

Finally, we need to check whether the user has pressed the button quickly enough (if at all). That means comparing the elapsed time, as measured by the 8 ms counter, with some threshold value – in this case 25, corresponding to a reaction time of 200 ms. The user has been successful if the 8 ms count is less than 25.

The easiest way to compare the magnitude of two values (is one larger or smaller than the other?) is to subtract them, and see if a *borrow* results.

If $A \geq B$, $A - B$ is positive or zero and no borrow is needed.

If $A < B$, $A - B$ is negative, requiring a borrow.

The baseline PICs provide just a single instruction for subtraction: ‘`subwf f,d`’ – “**subtract W** from **file register**”, where ‘`f`’ is the register being subtracted from, and, ‘`d`’ is the destination; ‘`f`’ to write the result back to the register, or ‘`w`’ to place the result in W.

The result of the subtraction is reflected in the Z (zero) and C (carry) bits in the STATUS register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	GPWUF	-	PA0	$\overline{\text{TO}}$	$\overline{\text{PD}}$	Z	DC	C

The Z bit is set if and only if the result is zero (so subtraction is another way to test for equality).

Although the C bit is called “carry”, in a subtraction it acts as a “not borrow”. That is, it is set to ‘1’ only if a borrow did *not* occur.

The table at the right shows the possible status flag outcomes from the subtraction $A - B$:

	Z	C
$A > B$	0	1
$A = B$	1	1
$A < B$	0	0

We can use this to test whether the elapsed time is less than 200 ms ($\text{cnt_8ms} < 25$) as follows:

```

movlw  .25          ; if time < 200 ms (25 x 8 ms)
subwf  cnt_8ms,w
btfss  STATUS,C
bsf    GPIO,1      ; turn on success LED

```

The subtraction being performed here is `cnt_8ms - 25`, so `C = 0` only if `cnt_8ms < 25` (see the table above). If `C = 1`, the elapsed time must be greater than the allowed 200 ms, and the instruction to turn on the success LED is skipped.

Complete program

Here's the complete code for the reaction timer, showing how the above code fragments fit together:

```

;*****
; Description: Lesson 5, example 1 *
; Reaction Timer game. *
; *
; Demonstrates use of Timer0 to time real-world events *
; *
; User must attempt to press button within 200 ms of "start" LED *
; lighting. If and only if successful, "success" LED is lit. *
; *
; Starts with both LEDs unlit. *
; 2 sec delay before lighting "start" *
; Waits up to 1 sec for button press *
; (only) on button press, lights "success" *
; 1 sec delay before repeating from start *
; *
;*****
; Pin assignments: *
; GP1 = success LED *
; GP2 = start LED *
; GP3 = pushbutton switch (active low) *
;*****
list p=12F509
#include <p12F509.inc>

EXTERN delay10_R ; W x 10 ms delay

;***** CONFIGURATION
; int reset, no code protect, no watchdog, int RC clock
__CONFIG _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC

;***** VARIABLE DEFINITIONS
UDATA
cnt_8ms res 1 ; counter: increments every 8 ms

;***** RC CALIBRATION
RCCAL CODE 0x3FF ; processor reset vector
res 1 ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET CODE 0x000 ; effective reset vector
movwf OSCCAL ; apply internal RC factory calibration
pagesel start
goto start ; jump to main code

;***** Subroutine vectors
delay10 ; delay W x 10 ms
pagesel delay10_R
goto delay10_R

;***** MAIN PROGRAM *****
MAIN CODE

```

```

;***** Initialisation
start
    ; configure ports
    movlw  b'111001'      ; configure GP1 and GP2 (only) as outputs
    tris   GPIO
    ; configure timer
    movlw  b'11010100'   ; configure Timer0:
                        ; --0-----   timer mode (T0CS = 0)
                        ; ----0----   prescaler assigned to Timer0 (PSA = 0)
                        ; -----100   prescale = 32 (PS = 100)
    option ; -> increment every 32 us

;***** Main loop
main_loop
    ; turn off both LEDs
    clrf   GPIO

    ; delay 2 sec
    movlw  .200           ; 200 x 10 ms = 2 sec
    pagesel delay10
    call   delay10
    pagesel $

    ; indicate start
    bsf    GPIO,2         ; turn on start LED

    ; wait up to 1 sec for button press
    banksel cnt_8ms      ; clear timer (8 ms counter)
    clrf   cnt_8ms       ; repeat for 1 sec:
waitls    clrf   TMR0    ; clear Timer0
w_tmr0    ; repeat for 8 ms:
    btfss  GPIO,3       ; if button pressed (GP3 low)
    goto   waitls_end   ; finish delay loop immediately
    movf   TMR0,w
    xorlw  .250         ; (250 ticks x 32 us/tick = 8 ms)
    btfss  STATUS,Z
    goto   w_tmr0
    incf   cnt_8ms,f    ; increment 8 ms counter
    movlw  .125         ; (125 x 8 ms = 1 sec)
    xorwf  cnt_8ms,w
    btfss  STATUS,Z
    goto   waitls

waitls_end

    ; indicate success if elapsed time < 200 ms
    movlw  .25          ; if time < 200 ms (25 x 8 ms)
    subwf  cnt_8ms,w
    btfss  STATUS,C
    bsf    GPIO,1       ; turn on success LED

    ; delay 1 sec
    movlw  .100         ; 100 x 10 ms = 1 sec
    pagesel delay10
    call   delay10
    pagesel $

    ; repeat forever
    goto   main_loop

END

```


Example 2: Flash LED while responding to input

As discussed above, timers can be used to maintain the accurate timing of regular (“background”) events, while performing other actions in response to input signals. To illustrate this, we’ll flash the LED on GP2 at 1 Hz (similar to [lesson 2](#)), while lighting the LED on GP1 whenever the pushbutton on GP3 is pressed (as was done in [lesson 4](#)). This example also shows how Timer0 can be used to provide a fixed delay.

When creating an application which performs a number of tasks, it is best, if practical, to implement and test each of those tasks separately. In other words, build the application a piece at a time, adding each new part to base that is known to be working. So we’ll start by simply flashing the LED.

The delay needs to be written in such a way that button scanning code can be added within it later. Calling a delay subroutine, as was done in [lesson 3](#), wouldn’t be appropriate; if the button press was only checked at the start and/or end of the delay, the button would seem unresponsive (a 0.5 sec delay is very noticeable).

Since the maximum delay that Timer0 can generate directly from a 1 MHz instruction clock is 65 ms, we have to extend the timer by adding a counter variable, as we did in example 1.

To produce a given delay, various combinations of prescaler value, maximum timer count and number of repetitions will be possible. But noting that $125 \times 125 \times 32 \mu\text{s} = 500 \text{ ms}$, a delay of exactly 500 ms can be generated by:

- Using a 4 MHz processor clock, providing a 1 MHz instruction clock and a 1 μs instruction cycle
- Assigning a 1:32 prescaler to the instruction clock, incrementing Timer0 every 32 μs
- Resetting Timer0 to zero, as soon as it reaches 125 (i.e. every $125 \times 32 \mu\text{s} = 4 \text{ ms}$)
- Repeating 125 times, creating a delay of $125 \times 4 \text{ ms} = 500 \text{ ms}$.

The following code implements the above steps:

```

;***** Initialisation
start
    ; configure ports
    clrf    GPIO           ; start with all LEDs off
    clrf    sGPIO         ; update shadow
    movlw   b'111001'     ; configure GP1 and GP2 (only) as outputs
    tris    GPIO
    ; configure timer
    movlw   b'11010100'   ; configure Timer0:
                        ; --0-----   timer mode (T0CS = 0)
                        ; ----0---    prescaler assigned to Timer0 (PSA = 0)
                        ; -----100   prescale = 32 (PS = 100)
    option          ; -> increment every 32 us

;***** Main loop
main_loop
    ; delay 500 ms
    banksel dly_cnt
    movlw   .125          ; repeat 125 times (125 x 4 ms = 500 ms)
    movwf   dly_cnt
dly500    clrf    TMR0     ; clear timer0
w_tmr0    movf    TMR0,w   ; wait for 4 ms
          xorlw   .125     ; (125 ticks x 32 us/tick = 4 ms)
          btfs   STATUS,Z
          goto   w_tmr0
          decfsz dlycnt,f  ; end 500 ms delay loop
          goto   dly500

    ; toggle flashing LED
    movf    sGPIO,w

```

```

xorlw    b'000100'      ; toggle LED on GP2
movwf   sGPIO          ; using shadow register
movwf   GPIO

; repeat forever
goto    main_loop

```

Here's the code developed in [lesson 4](#), for turning on a LED when the pushbutton is pressed:

```

clrf    sGPIO          ; assume button up -> LED off
btfss   GPIO,3        ; if button pressed (GP3 low)
bsf     sGPIO,1       ; turn on LED

movf    sGPIO,w       ; copy shadow to GPIO
movwf   GPIO

```

It's quite straightforward to place some code similar to this (replacing the `clrf` with a `bcf` instruction, to avoid affecting any other bits in the shadow register) within the timer wait loop; since the timer increments every 32 instructions, there are plenty of cycles available to accommodate these additional instructions, without risk that the "TMRO = 125" condition will be skipped (see discussion in example 1).

Here's how:

```

w_tmr0          ; repeat for 4 ms:
                ; check and respond to button press
                ; assume button up -> indicator LED off
bcf        sGPIO,1      ; if button pressed (GP3 low)
btfss     GPIO,3        ; turn on indicator LED
bsf       sGPIO,1       ; update port (copy shadow to GPIO)
movf      sGPIO,w
movwf     GPIO
movf      TMR0,w
xorlw    .125          ; (125 ticks x 32 us/tick = 4 ms)
btfss    STATUS,Z
goto     w_tmr0

```

Complete program

Here's the complete code for the flash + pushbutton demo.

Note that, because GPIO is being updated from the shadow copy on every "spin" of the timer wait loop, there is no need to update GPIO when the LED on GP2 is toggled; the change will be picked up next time through the timer wait loop.

```

;*****
;
; Description: Lesson 5, example 2
;
; Demonstrates use of Timer0 to maintain timing of background actions
; while performing other actions in response to changing inputs
;
; One LED simply flashes at 1 Hz (50% duty cycle).
; The other LED is only lit when the pushbutton is pressed
;
;*****
;
; Pin assignments:
; GP1 = "button pressed" indicator LED
; GP2 = flashing LED
; GP3 = pushbutton switch (active low)
;
;*****

```

```

list      p=12F509
#include   <p12F509.inc>

;***** CONFIGURATION
                ; int reset, no code protect, no watchdog, int RC clock
__CONFIG     _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC

;***** VARIABLE DEFINITIONS
                UDATA_SHR
sGPIO        res 1                ; shadow copy of GPIO

                UDATA
dly_cnt      res 1                ; delay counter

;***** RC CALIBRATION
RCCAL        CODE    0x3FF        ; processor reset vector
                res 1            ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET        CODE    0x000        ; effective reset vector
                movwf    OSCCAL    ; apply internal RC factory calibration

;***** MAIN PROGRAM *****

;***** Initialisation
start
                ; configure ports
                clrf     GPIO      ; start with all LEDs off
                clrf     sGPIO     ; update shadow
                movlw   b'111001'  ; configure GP1 and GP2 (only) as outputs
                tris    GPIO
                ; configure timer
                movlw   b'11010100' ; configure Timer0:
                ; --0-----   timer mode (T0CS = 0)
                ; ----0---    prescaler assigned to Timer0 (PSA = 0)
                ; -----100   prescale = 32 (PS = 100)
                option    ; -> increment every 32 us

;***** Main loop
main_loop
                ; delay 500 ms while responding to button press
                banksel dly_cnt
                movlw   .125        ; repeat 125 times (125 x 4 ms = 500 ms)
                movwf   dly_cnt
dly500        clrf     TMR0        ; clear timer0
w_tmr0       ; repeat for 4 ms:
                ; check and respond to button press
                bcf     sGPIO,1    ; assume button up -> indicator LED off
                btfs   GPIO,3     ; if button pressed (GP3 low)
                bsf     sGPIO,1    ; turn on indicator LED
                movf    sGPIO,w    ; update port (copy shadow to GPIO)
                movwf   GPIO
                movf    TMR0,w
                xorlw   .125        ; (125 ticks x 32 us/tick = 4 ms)
                btfs   STATUS,Z
                goto    w_tmr0

```

```

    decfsz  dly_cnt,f           ; end 500 ms delay loop
    goto   dly500

    ; toggle flashing LED
    movf   sGPIO,w
    xorlw  b'000100'          ; toggle LED on GP2
    movwf  sGPIO              ; using shadow register

    ; repeat forever
    goto  main_loop

END

```

Example 3: Switch debouncing

[Lesson 4](#) explored the topic of switch bounce, and described a counting algorithm to address it, which was expressed as:

```

count = 0
while count < max_samples
    delay sample_time
    if input = required_state
        count = count + 1
    else
        count = 0
end

```

The switch is deemed to have changed when it has been continuously in the new state for some minimum period, for example 10 ms. This is done by continuing to increment a count while checking the state of the switch. “Continuing to increment a count” while something else (such as checking a switch) occurs is exactly what a timer does. Since a timer increments automatically, using a timer can simplify the logic, as follows:

```

reset timer
while timer < debounce time
    if input ≠ required_state
        reset timer
end

```

On completion, the input will have been in the required state (changed) for the minimum debounce time.

Assuming a 1 MHz instruction clock and a 1:64 prescaler, a 10 ms debounce time will be reached when the timer reaches $10\text{ ms} \div 64\ \mu\text{s} = 156.3$; taking the next highest integer gives 157.

The following code demonstrates how Timer0 can be used to debounce a “button down” event:

```

wait_dn  clrf    TMR0           ; reset timer
chk_dn   btfsc  GPIO,3         ; check for button press (GP3 low)
         goto   wait_dn        ; continue to reset timer until button down
         movf   TMR0,w         ; has 10ms debounce time elapsed?
         xorlw  .157           ; (157 = 10ms/64us)
         btfss  STATUS,Z       ; if not, continue checking button
         goto  chk_dn

```

That’s shorter than the equivalent routine presented in [lesson 4](#), and it avoids the need to use two data registers as counters. But – it uses Timer0, and on baseline PICs, there is only one timer. It’s a scarce resource! If you’re using it to time a regular background process, as we did in example 2, you won’t be able

to use it for debouncing. You must be careful, as you build a library of routines that use Timer0, if you use more than one routine which uses Timer0 in a program, that the way they use or setup Timer0 doesn't clash. But if you're not using Timer0 for anything else, using it for switch debouncing is perfectly reasonable.

Complete program

The following program is equivalent to that presented in lesson 4. By using Timer0 for debouncing, it's shorter and uses less data memory:

```

;*****
; Description: Lesson 5, example 3 *
; *
; Demonstrates use of Timer0 to implement debounce counting algorithm *
; *
; Toggles LED when pushbutton is pressed then released *
; *
;*****
; Pin assignments: *
; GP1 = indicator LED *
; GP3 = pushbutton switch (active low) *
; *
;*****

list p=12F509
#include <p12F509.inc>

;***** CONFIGURATION
; int reset, no code protect, no watchdog, int RC clock
__CONFIG _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntrC_OSC

;***** VARIABLE DEFINITIONS
UDATA_SHR
sGPIO res 1 ; shadow copy of GPIO

;***** RC CALIBRATION
RCCAL CODE 0x3FF ; processor reset vector
res 1 ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET CODE 0x000 ; effective reset vector
movwf OSCCAL ; apply internal RC factory calibration

;***** MAIN PROGRAM *****

;***** Initialisation
start
; configure ports
clrf GPIO ; start with LED off
clrf sGPIO ; update shadow
movlw b'111101' ; configure GP1 (only) as an output
tris GPIO
; configure timer
movlw b'11010101' ; configure Timer0:
; --0----- timer mode (T0CS = 0)
; ----0--- prescaler assigned to Timer0 (PSA = 0)
; -----101 prescale = 64 (PS = 101)
option ; -> increment every 64 us

```

```

;***** Main loop
main_loop
    ; wait for button press, debounce using timer0:
wait_dn  clrf    TMR0            ; reset timer
chk_dn   btfs   GPIO,3         ; check for button press (GP3 low)
        goto   wait_dn        ; continue to reset timer until button down
        movf   TMR0,w         ; has 10ms debounce time elapsed?
        xorlw  .157          ; (157=10ms/64us)
        btfs   STATUS,Z       ; if not, continue checking button
        goto   chk_dn

    ; toggle LED on GP1
    movf    sGPIO,w
    xorlw   b'000010'         ; toggle shadow register
    movwf   sGPIO
    movwf   GPIO             ; write to port

    ; wait for button release, debounce using timer0:
wait_up  clrf    TMR0            ; reset timer
chk_up   btfs   GPIO,3         ; check for button release (GP3 high)
        goto   wait_up        ; continue to reset timer until button up
        movf   TMR0,w         ; has 10ms debounce time elapsed?
        xorlw  .157          ; (157=10ms/64us)
        btfs   STATUS,Z       ; if not, continue checking button
        goto   chk_up

    ; repeat forever
    goto   main_loop

END

```

Counter Mode

As explained above, Timer0 can also be used to count external events, consisting of a transition (rising or falling) on the T0CKI input.

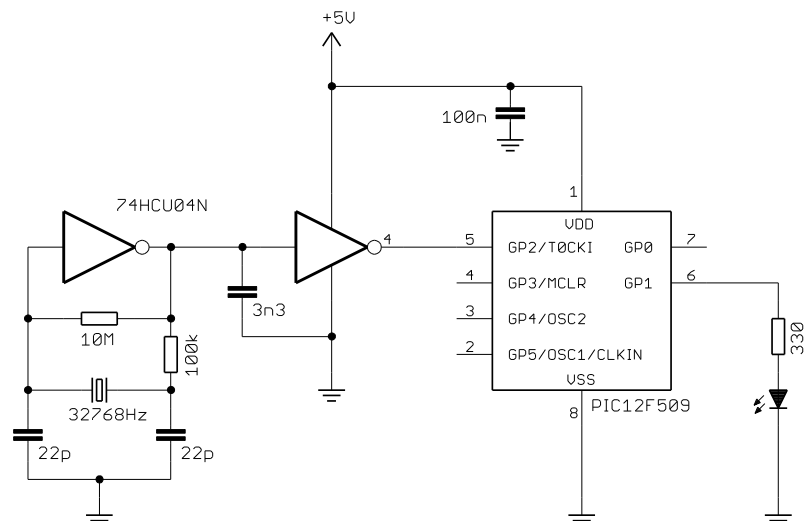
This is useful in a number of ways, such as performing an action after some number of input transitions, or measuring the frequency of an input signal, for example from a sensor triggered by the rotation of an axle. The frequency in Hertz of the signal is simply the number of transitions counted in 1 s.

However, it's not really practical to build a frequency counter, using only the techniques (and microcontrollers) we've covered so far!

To show how to use Timer0 as a counter, we'll go back to LED flashing, but driving the counter with a crystal-based external clock, providing a much more accurate time base.

The circuit used for this is as shown on the right.

A 32.768 kHz "watch crystal" is driven by a CMOS inverter to generate a 32.768 kHz clock signal.

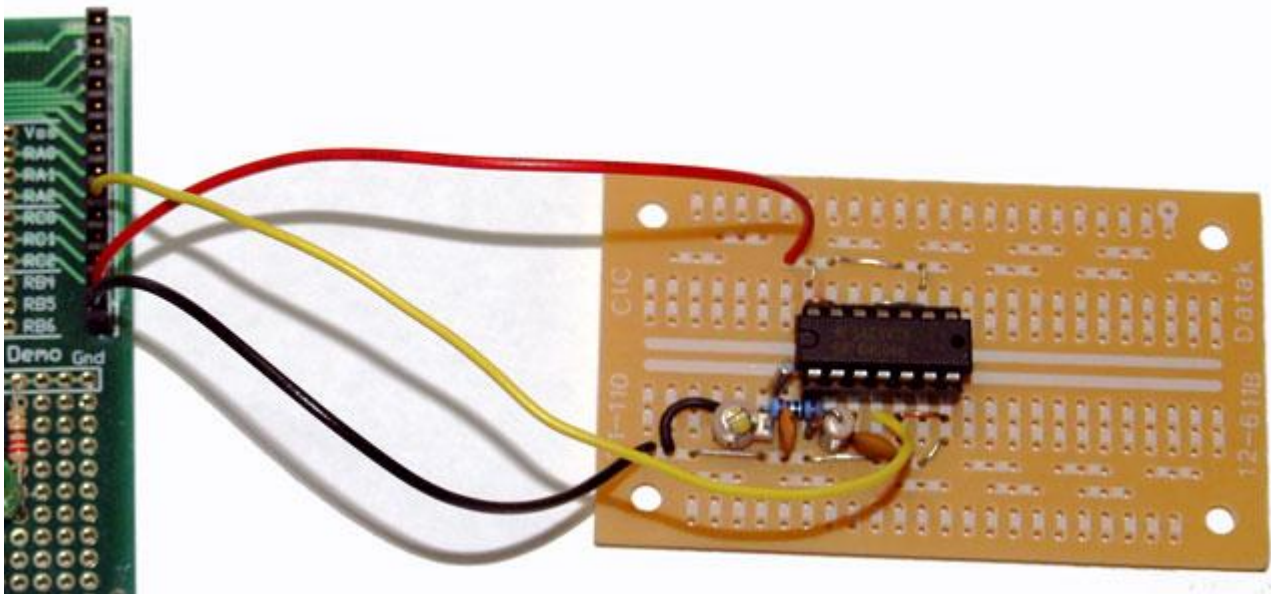


The capacitor and resistor values shown should work with most watch crystals designed for a 12.5 pF load capacitance. The exact value of the capacitor loading the first inverter (3.3 nF here) isn't important – and in theory isn't necessary – but in practice it helps the oscillator start reliably.

The oscillator output is buffered by another inverter, before being fed to the T0CKI (GP2) input on the PIC.

The [Gooligum baseline training board](#) already has this oscillator circuit in place (in the upper right of the board) – jumper JP22 connects the 32 kHz clock signal to T0CKI. And, as before, jumper JP12 enables the LED on GP1.

If you have Microchip's Low Pin Count Demo Board, you will need to build the oscillator circuit separately. Since it will be used in a number of lessons, and to limit the effect of stray capacitance, it's a good idea to build it on something more permanent than breadboard, such as a strip prototyping board. You should then connect it to the 14-pin header on the demo board (GP2/T0CKI is brought out as pin 9 on the header, while power and ground are pins 13 and 14), as illustrated in the photograph below:



Note that in this photo, one of the load capacitors is variable – this makes it possible to tune the oscillation frequency, but it's not really necessary.

We'll use this 32.768 kHz signal with Timer0, to generate the timing needed to flash the LED on GP1 at almost exactly 1 Hz (the accuracy being set by the accuracy of the crystal oscillator, which can be expected to be much better than that of the PIC's internal RC oscillator).

Those familiar with binary numbers will have noticed that $32768 = 2^{15}$, making it very straightforward to divide the 32768 Hz input down to 1 Hz.

Since $32768 = 128 \times 256$, if we apply a 1:128 prescale ratio to the 32768 Hz signal on T0CKI, TMR0 will be incremented 256 times per second. The most significant bit of TMR0 (TMR0<7>) will therefore be cycling at a rate of exactly 1 Hz; it will be '0' for 0.5 s, followed by '1' for 0.5 s.

So if we clock TMR0 with the 32768 Hz signal on T0CKI, prescaled by 128, the task is simply to light the LED (GP1 high) when TMR0<7> = 1, and turn off the LED (GP1 low) when TMR0<7> = 0.

To configure Timer0 for counter mode (external clock on TOCKI) with a 1:128 prescale ratio, set the T0CS bit to '1', PSA to '0' and PS<2:0> to '110':

```

movlw    b'11110110'    ; configure Timer0:
          ; --1-----    counter mode (T0CS = 1)
          ; ----0----    prescaler assigned to Timer0 (PSA = 0)
          ; -----110    prescale = 128 (PS = 110)
option   ; -> increment at 256 Hz with 32.768 kHz input

```

Note that the value of T0SE bit is irrelevant; we don't care if the counter increments on the rising or falling edge of the signal on TOCKI – only the frequency is important. Either edge will do.

Next we need to continually set GP1 high whenever TMR0<7> = 1, and low whenever TMR0<7> = 0.

In other words, continually update GP1 with the current value of TMR0<7>.

Unfortunately, there is no simple “copy a single bit” instruction in baseline PIC assembler!

If you're not using a shadow register for GPIO, the following “direct approach” is effective, if a little inelegant:

```

start    ; transfer TMR0<7> to GP1
         btfsc   TMR0,7          ; if TMR0<7>=1
         bsf     GPIO,1         ; set GP1
         btfss   TMR0,7          ; if TMR0<7>=0
         bcf     GPIO,1         ; clear GP1

         ; repeat forever
         goto    start

```

As described in [lesson 4](#), if you are using a shadow register (as previously discussed, it's generally a good idea to do so, to avoid potential, and difficult to debug, problems), this can be implemented as:

```

loop     ; transfer TMR0<7> to GP1
         clrfs   sGPIO          ; assume TMR0<7>=0 -> LED off
         btfsc   TMR0,7          ; if TMR0<7>=1
         bsf     sGPIO,1        ; turn on LED

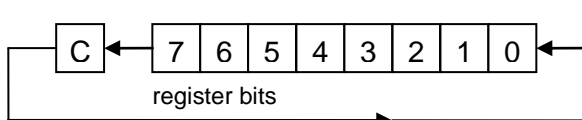
         movf    sGPIO,w        ; copy shadow to GPIO
         movwf   GPIO

         ; repeat forever
         goto    loop

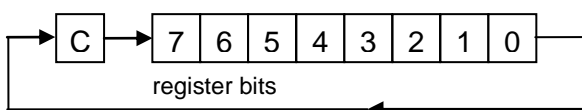
```

But since this is actually an instruction longer, it's only really simpler if you were going to use a shadow register anyway.

Another approach is to use the PIC's rotate instructions. These instructions move every bit in a register to the left or right, as illustrated:



'rlf f,d' – “rotate left file register through carry”



'rrf f,d' – “rotate right file register through carry”

In both cases, the bit being rotated out of bit 7 (for `rlf`) or bit 0 (for `rrf`) is copied into the carry bit in the `STATUS` register, and the previous value of carry is rotated into bit 0 (for `rlf`) or bit 7 (for `rrf`).

As usual, ‘`r`’ is the register being rotated, and ‘`d`’ is the destination: ‘`,r`’ to write the result back to the register, or ‘`,w`’ to place the result in `W`.

The ability to place the result in `W` is useful, since it means that we can “left rotate” `TMR0`, to copy the current value from `TMR0<7>` into `C`, without affecting the value in `TMR0`.

There are no instructions for rotating `W`, only the addressable special-function and general purpose registers. That’s a pity, since such an instruction would be useful here. Instead, we’ll need to rotate the bit copied from `TMR0<7>` into bit 0 of a temporary register, then rotate again to move the copied bit into bit 1, and then copy the result to `GPIO`, as follows:

```

rlf    TMR0,w          ; copy TMR0<7> to C
clrf   temp           ; clear temp
rlf    temp,f         ; rotate C into temp
rlf    temp,w         ; rotate once more into W (-> W<1> = TMR0<7>)
movwf  GPIO           ; update GPIO with result (-> GP1 = TMR0<7>)

```

Note that ‘`temp`’ is cleared before being used. That’s not strictly necessary in this example; since only `GP1` is being used as an output, it doesn’t actually matter what the other bits in `GPIO` are set to.

Of course, if any other bits in `GPIO` were being used as outputs, you couldn’t use this method anyway, since this code will clear every bit other than `GP1`! In that case, you’re better off using the bit test and set/clear instructions, which are generally the most practical way to “copy a bit”. But it’s worth remembering that the rotate instructions are also available, and using them may lead to shorter code.

Complete program

Here’s the complete “flash a LED at 1 Hz using a crystal oscillator” program, using the “copy a bit via rotation” method:

```

;*****
;
; Description:      Lesson 5, example 4b
;
; Demonstrates use of Timer0 in counter mode and rotate instructions
;
; LED flashes at 1 Hz (50% duty cycle),
; with timing derived from 32.768 kHz input on T0CKI
;
; Uses rotate instructions to copy MSB from Timer0 to GP1
;
;*****
;
; Pin assignments:
; GP1      = flashing LED
; T0CKI    = 32.768 kHz signal
;
;*****

list          p=12F509
#include      <p12F509.inc>

;***** CONFIGURATION
; ext reset, no code protect, no watchdog, int RC clock
__CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC

```

```

;***** VARIABLE DEFINITIONS
        UDATA_SHR
temp    res 1                ; temp register used for rotates

;***** RC CALIBRATION
RCCAL   CODE    0x3FF        ; processor reset vector
        res 1                ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET   CODE    0x000        ; effective reset vector
        movwf   OSCCAL       ; apply internal RC factory calibration

;***** MAIN PROGRAM *****

;***** Initialisation
start
        ; configure port
        movlw   b'111101'    ; configure GP1 (only) as output
        tris    GPIO
        ; configure timer
        movlw   b'11110110'  ; configure Timer0:
                                ; --1-----   counter mode (T0CS = 1)
                                ; ----0---     prescaler assigned to Timer0 (PSA = 0)
                                ; -----110    prescale = 128 (PS = 110)
        option  ; -> increment at 256 Hz with 32.768 kHz input

;***** Main loop
main_loop
        ; TMR0<7> cycles at 1Hz, so continually copy to LED (GP1)
        rlf     TMR0,w        ; copy TMR0<7> to C
        clrf    temp
        rlf     temp,f        ; rotate C into temp
        rlf     temp,w        ; rotate once more into W (-> W<1> = TMR0<7>)
        movwf   GPIO         ; update GPIO with result (-> GP1 = TMR0<7>)

        ; repeat forever
        goto    main_loop

        END

```

Conclusion

Hopefully the examples in this lesson have given you an idea of the flexibility and usefulness of the Timer0 peripheral.

With it, we were able to:

- Time an event
- Perform a periodic action while responding to input
- Debounce a switch
- Count external pulses

We'll revisit Timer0 later, and introduce other timers when we move onto the mid-range architecture.

But first, in the [next lesson](#) we'll take a quick look at how some of the MPASM assembler directives can be used to make our code easier to read and maintain, which will be important as our programs grow bigger.

Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 6: Assembler Directives and Macros

As the programs presented in these tutorials are becoming longer, it's appropriate to take a look at some of the facilities that MPASM (the Microchip PIC assembler) provides to simplify the process of writing and maintaining code.

This lesson covers:

- Arithmetic and bitwise operators
- Text substitution with `#define`
- Defining constants with `equ` or `constant`
- Conditional assembly using `if / else / endif`, `ifdef` and `ifndef`
- Outputting warning and error messages
- Assembler macros

Each of these topics is illustrated by making use of it in code from previous lessons in this series.

Arithmetic Operators

MPASM supports the following arithmetic operators:

negate	-
multiply	*
divide	/
modulus	%
add	+
subtract	-

Precedence is in the traditional order, as above.

For example, $2 + 3 * 4 = 2 + 12 = 14$.

To change the order of precedence, use parentheses: (and).

For example, $(2 + 3) * 4 = 5 * 4 = 20$.

*Note: These calculations take place during the assembly process, before any code is generated. They are used to calculate constant values which will be included in the code to be assembled. They **do not** generate any PIC instructions.*

These arithmetic operators are useful in showing how a value has been derived, making it easier to understand the code and to make changes.

For example, consider this code from [lesson 2](#):

```

; delay 500 ms
movlw    .244           ; outer loop: 244 x (1023 + 1023 + 3) + 2
movwf    dc2           ;   = 499,958 cycles
clrf     dc1           ; inner loop: 256 x 4 - 1
dly1     nop           ; inner loop 1 = 1023 cycles
         decfsz    dc1,f
         goto     dly1
dly2     nop           ; inner loop 2 = 1023 cycles
         decfsz    dc1,f
         goto     dly2
         decfsz    dc2,f
         goto     dly1

```

Where does the value of 244 come from? It is the number of outer loop iterations needed to make 500 ms.

To make this clearer, we could change the comments to:

```

; delay 500 ms
movlw    .244           ; outer loop: #iterations =
movwf    dc2           ;   500ms/(1023+1023+3)us/loop = 244

```

Or, instead of writing the constant '244' directly, write it as an expression:

```

; delay 500 ms
movlw    .500000/(.1023+.1023+.3) ; number of outer loop iterations
movwf    dc2                 ; for 500 ms delay

```

If you're using mainly decimal values in your expressions, as here, you may wish to change the default radix to decimal, to avoid having to add a '.' before each decimal value.

That's not necessarily a good idea; if your code assumes that some particular default radix has been set, you need to be very careful if you copy that code into another program, which may have a different default radix. But, if you're prepared to take the risk, add the 'radix' directive near the start of the program.

For example:

```
radix    dec
```

The valid radix values are 'hex' for hexadecimal (base 16), 'dec' for decimal (base 10) and 'oct' for octal (base 8). The default radix is hex.

With the default radix set to decimal, this code fragment can be written as:

```

; delay 500 ms
movlw    500000/(1023+1023+3) ; # outer loop iterations for 500 ms
movwf    dc2

```

Defining Constants

Programs often contain numeric values which may need to be tuned or changed later, particularly during development. When a change needs to be made, finding these values in the code can be difficult. And making changes may be error-prone if the same value (or another value derived from the value being changed) occurs more than once in the code.

To make the code more maintainable, each constant value should be defined only once, near the start of the program, where it is easy to find and change.

A good example is the reaction timer developed in [lesson 5](#), where “success” was defined as pressing a pushbutton less than 200 ms after a LED was lit. But what if, during testing, we found that 200 ms is unrealistically short? Or too long?

To change this maximum reaction time, you’d need to find and then modify this fragment of code:

```

; indicate success if elapsed time < 200 ms
movlw    .25                ; if time < 200 ms (25 x 8 ms)
subwf    cnt_8ms,w
btfss    STATUS,C
bsf      GPIO,1            ; turn on success LED

```

To make this easier to maintain, we could define the maximum reaction time as a constant, at the start of the program.

This can be done using the ‘equ’ (short for “equate”) directive, as follows:

```
MAXRT    equ    .200        ; Maximum reaction time in ms
```

Alternatively, you could use the ‘constant’ directive:

```
constant MAXRT=.200        ; Maximum reaction time in ms
```

The two directives are equivalent. Which you choose to use is simply a matter of style.

‘equ’ is more commonly found in assemblers, and perhaps because it is more familiar, most people use it.

Personally, I prefer to use ‘constant’, mainly because I like to think of any symbol placed on the left hand edge (column 1) of the assembler source as being a label for a program or data register address, and I prefer to differentiate between address labels and constants to be used in expressions. But it’s purely your choice.

However you define this constant, it can be referred to later in your code, for example:

```

; check elapsed time
movlw    MAXRT/8           ; if time < max reaction time (8 ms/count)
subwf    cnt_8ms,w
btfss    STATUS,C
bsf      GPIO,1            ; turn on success LED

```

Note how constants can be usefully included in arithmetic expressions. In this way, the constant can be defined simply in terms of real-world quantities (e.g. ms), making it readily apparent how to change it to a new value (e.g. 300 ms), while arithmetic expressions are used to convert that into a quantity that matches the program’s logic. And if that logic changes later (say, counting by 16 ms instead of 8 ms increments), then only the arithmetic expression needs to change; the constant can remain defined in the same way.

Text Substitution

As well as being able to define numeric constants, it is also very useful to be able to define “text constants”, where a text *string* is substituted into the assembler source code.

Text substitution is commonly used to refer to I/O pins by a descriptive label. This makes your code more readable, and easier to update if pin assignments change later.

Why would pin assignments change? Whether you design your own printed circuit boards, or layout your circuit on prototyping board, swapping pins around can often simplify the physical circuit layout. That’s one of the great advantages of designing with microcontrollers; as you layout your design, you can go back and modify the code to simplify that layout, perhaps repeating that process a number of times.

For example, consider again the reaction timer from [lesson 5](#). The I/O pins were assigned as follows:

```
; Pin assignments:
; GP1 = success LED
; GP2 = start LED
; GP3 = pushbutton
```

These assignments are completely arbitrary; the LEDs could be on any pin other than GP3 (which is input only), while the pushbutton could be on any unused pin.

One way of defining these pins would be to use numeric constants:

```
constant nSTART=2      ; start LED
constant nSUCCESS=1    ; success LED
constant nBUTTON=3     ; pushbutton
```

(The ‘n’ prefix used here indicates that these are numeric constants; this is simply a convention, and you can choose whatever naming style works for you.)

They would then be referenced in the code, as for example:

```
        bsf      GPIO,nSTART      ; turn on start LED
w_tmr0  btfss    GPIO,nBUTTON     ; check for button press (low)
        bsf      GPIO,nSUCCESS    ; turn on success LED
```

A significant problem with this approach is that larger PICs (i.e. most of them!) have more than one port. Instead of GPIO, larger PICs have ports named PORTA, PORTB, PORTC, and so on. What if you moved an input or output from PORTA to PORTC? The above approach, using numeric constants, wouldn’t work, because you’d have to go through your code and change all the PORTA references to PORTC.

This problem can be solved using text substitution, using the ‘#define’ directive, as follows:

```
#define START      GPIO,2      ; start LED
#define SUCCESS    GPIO,1      ; success LED
#define BUTTON     GPIO,3      ; pushbutton
```

These definitions are then referenced later in the code, as shown:

```
        bsf      START           ; turn on start LED
w_tmr0  btfss    BUTTON         ; check for button press (low)
        bsf      SUCCESS        ; turn on success LED
```

Note that there are no longer any references to GPIO in the main body of the code. If you later move this code to a PIC with more ports, you only need to update the definitions at the start. Of course, you would also need to modify the corresponding port initialisation code, such as ‘tris’ instructions. This is a good reason to keep all your initialisation code in one easily-found place, such as at the start of the program, or in an “init” subroutine.

Bitwise Operators

We’ve seen that operations on binary values are fundamental to PIC microcontrollers: setting and clearing individual bits, flipping bits, testing the status of bits and rotating the bits in registers. Many configuration options are specified as a collection of bits (or flags) which are assembled into a byte or word; for example, the ‘__CONFIG’ directive and the ‘option’ instruction.

To facilitate operations on bits, MPASM provides the following bitwise operators:

compliment	~
left shift	<<
right shift	>>
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	

Precedence is in the order listed above.

Parentheses are used to change the order of precedence: ‘(’ and ‘)’.

We’ve seen an example of the bitwise AND operator being used in every program so far:

```

                ; int reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG      _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC

```

These symbols are defined in the ‘p12F509.inc’ include file as follows:

```

;----- CONFIG Options -----
_OSC_LP            EQU  H'0FFC'      ; LP oscillator
_LP_OSC           EQU  H'0FFC'      ; LP oscillator
_OSC_XT           EQU  H'0FFD'      ; XT oscillator
_XT_OSC           EQU  H'0FFD'      ; XT oscillator
_OSC_IntRC        EQU  H'0FFE'      ; internal RC oscillator
_IntRC_OSC        EQU  H'0FFE'      ; internal RC oscillator
_OSC_ExtRC        EQU  H'0FFF'      ; external RC oscillator
_ExtRC_OSC        EQU  H'0FFF'      ; external RC oscillator
_WDT_OFF          EQU  H'0FFB'      ; WDT disabled
_WDT_ON           EQU  H'0FFF'      ; WDT enabled
_CP_ON            EQU  H'0FF7'      ; Code protection on
_CP_OFF           EQU  H'0FFF'      ; Code protection off
_MCLRE_OFF        EQU  H'0FEF'      ; GP3/MCLR pin function is digital input
_MCLRE_ON         EQU  H'0FFF'      ; GP3/MCLR pin function is MCLR

```

The ‘equ’ directive is described above; you can see that these are simply symbols for numeric constants.

In binary, the values in the ‘__CONFIG’ directive above are:

```

_MCLRE_OFF        H'0FEF' = 1111 1110 1111
_CP_OFF           H'0FFF' = 1111 1111 1111
_WDT_OFF          H'0FFB' = 1111 1111 1011
_IntRC_OSC        H'0FFE' = 1111 1111 1110

```

ANDing these together gives: 1111 1110 1010

So the directive above is equivalent to:

```

__CONFIG      b'111111101010'

```

For each of these configuration bit symbols, where a bit in the definition is ‘0’, it has the effect of setting the corresponding bit in the configuration word to ‘0’, because either ‘0’ or ‘1’ ANDed with ‘0’ equals ‘0’.

The 12-bit configuration word in the PIC12F509 is as shown:

Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	-	-	-	-	MCLRE	$\overline{\text{CP}}$	WDTE	FOSC1	FOSC0

These configuration options were described briefly in lessons [1](#) and [3](#). Recapping:

MCLR enables the external processor reset, or “master clear”, on pin 4. Clearing it allows GP3 to be used as an input.

$\overline{\text{CP}}$ disables code protection. Clearing $\overline{\text{CP}}$ protects your code from being read by PIC programmers.

WDTE enables the watchdog timer, which is used to reset the processor if it crashes. For more details, see [lesson 7](#). Clearing WDTE disables the watchdog timer.

The FOSC bits set the clock, or oscillator, configuration; FOSC<1:0> = 10 specifies the internal RC oscillator. The other oscillator configurations are described in [lesson 7](#).

Given this, to configure the PIC12F509 for internal reset (GP3 as an input), no code protection, no watchdog timer and the internal RC oscillator, the lower five bits of the configuration word must be set to 01010.

That’s the same pattern of bits produced by the `__CONFIG` directive, above (the value of the upper seven bits is irrelevant, as they are not used), showing that ANDing together the symbols in the Microchip-provided include file gives the correct result. Using the symbols is simpler, and safer; it’s easy to mistype a long binary value, leading to a difficult-to-debug processor configuration error. If you mistype a symbol, the assembler will tell you, making it easy to correct the mistake.

It is also useful (clearer and less error-prone) to be able to use symbols instead of binary numbers when setting bits in special-function registers, such as OPTION.

The bits in the OPTION register are defined in the ‘p12F509.inc’ include file as follows:

```
PSA           EQU   H'0003'
TOSE          EQU   H'0004'
TOCS          EQU   H'0005'
NOT_GPPU      EQU   H'0006'
NOT_GPWU      EQU   H'0007'
PS0           EQU   H'0000'
PS1           EQU   H'0001'
PS2           EQU   H'0002'
```

Unlike the definitions used for the configuration bits, these symbols define a bit *position*, not a pattern. It tells us, for example, that TOCS is bit 5.

To set only bit 5, normally we’d use something like:

```
movlw  b'00100000'      ; select counter mode: TOCS=1
option                               ; (set bit 5 in OPTION)
```

But note that the binary constant `b'00100000'` is a ‘1’ shifted left five times, and so can be represented by the expression “`1<<5`”, using the binary left-shift operator ‘<<’.

That means that this code can be used instead of the above:

```
movlw  1<<TOCS          ; select counter mode: TOCS=1
option
```

This works because the symbol ‘TOCS’ is defined in the include file to be equal to ‘5’.

This code is clearer, needing fewer comments, and it is harder to make a mistake, as mistyping a symbol is likely to be picked up by the assembler, while mistyping a binary constant (say getting the ‘1’ in the wrong position) is likely to be missed.

Typically a number of bits need to be set at the same time. To do this, simply bitwise-OR the expressions together. For example, the crystal-based LED flasher code from [lesson 5](#) included:

```
movlw    b'11110110'    ; configure Timer0:
                ; --1-----    counter mode (T0CS = 1)
                ; ----0----    prescaler assigned to Timer0 (PSA = 0)
                ; -----110    prescale = 128 (PS = 110)
option   ; -> increment at 256 Hz with 32.768 kHz input
```

This can be replaced by:

```
                ; configure Timer0
movlw    1<<T0CS|0<<PSA|b'110'
                ; counter mode (T0CS = 1)
                ; prescaler assigned to Timer0 (PSA = 0)
                ; prescale = 128 (PS = 110)
option   ; -> increment at 256 Hz with 32.768 kHz input
```

Including '0<<PSA' in the expression does nothing, since a zero left-shifted any number of times is still zero, and ORing zero into any expression has no effect. But it makes it explicit that we are clearing PSA.

In this application, we don't care what the $\overline{\text{GPWU}}$, $\overline{\text{GPPU}}$ and TOSE bits are set to, so they are not included in the expression. And where a bit field (such as $\text{PS}<2:0>$) is most clearly expressed as a binary pattern, it can be ORed into the expression as a binary constant, as shown.

On the other hand, the first form, where the value to be loaded into a register is presented as a binary number and comments are used to show what each bit position means, is often clearer, so that's the style we'll mainly use in these lessons. But, as in most questions of style, it's up to you! In some cases, the second style, where a number of expressions are ORed together is clearer, and you'll sometimes encounter it in other people's code – so it's useful to be able to understand expressions like '1<<T0CS|0<<PSA|b'110'' even if you don't use them yourself.

Macros

We've seen in [lesson 3](#) that, if we wish to reuse the same piece of code a number of times in a program, it often makes sense to place that code into a subroutine and to call the subroutine from the main program.

But that's not always appropriate, or even possible. The subroutine call and return is an overhead that takes some time; only four instruction cycles, but in timing-critical pieces of code, it may not be justifiable. A more significant problem is that baseline PICs have only two stack registers, meaning that you must be very careful when nesting subroutine calls, or else the stack will overflow and your subroutine won't return to the right place. It's usually not worth using up a stack level, just to avoid repeating a short piece of code.

Another problem with subroutines is that, as we saw in lesson 3, to pass parameters to them, you need to load the parameters into registers – an overhead that leads to longer code, perhaps negating the space-saving advantage of using a subroutine, for small pieces of code. And loading parameters into registers, before calling a subroutine, isn't very readable. It would be nicer to be able to simply list the parameters on a single line, as part of the subroutine call.

Macros address these problems, and are often appropriate where a subroutine is not. A *macro* is a sequence of instructions that is inserted (or *expanded*) into the source code by the assembler, prior to assembly.

*Note: The purpose of a macro is to make the source code more compact; unlike a subroutine, it **does not** make the resultant object code any smaller. The instructions within a macro are expanded into the source code, every time the macro is called.*

Here's a simple example. [Lesson 3](#) introduced a 'delay10' subroutine, which took as a parameter in W a delay as a multiples of 10 ms. So to delay for 200 ms, we had:

```
movlw    .20                ; delay 20 x 10 ms = 200 ms
call     delay10
```

This was used in a program which flashed a LED with a 20% duty cycle: on for 200 ms, then off for 800 ms. Rewritten a little from the code presented in lesson 3, the main loop looks like this:

```
main_loop
    bsf     FLASH            ; turn on LED
    movlw   .20              ; stay on for 200 ms
    pagesel delay10         ; (delay 20 x 10 ms)
    call    delay10
    bcf     FLASH            ; turn off LED
    movlw   .80              ; stay off for 800 sec
    call    delay10         ; (delay 80 x 10 ms)
    pagesel main_loop       ; repeat forever
    goto    main_loop
```

It would be nice to be able to simply write something like 'DelayMS 200' for a 200 ms delay.

We can do that by defining a macro, as follows:

```
DelayMS MACRO    ms                ; delay time in ms
    movlw   ms/.10                ; divide by 10 to pass to delay10 routine
    pagesel delay10
    call    delay10
    pagesel $
ENDM
```

This defines a macro called 'DelayMS', which takes a single parameter: 'ms', the delay time in milliseconds. Parameters are referred to within the macro in the same way as any other symbol, and can be used in expressions, as shown.

A macro definition consists of a label (the macro's name), the 'MACRO' directive, and a comma-separated list of symbols, or *arguments*, used to pass parameters to the macro, all on one line. It is followed by a sequence of instructions and/or assembler directives, finishing with the 'ENDM' directive.

When the source code is assembled, the macro's instruction sequence is inserted into the code, with the arguments replaced by the parameters that were passed to the macro.

That may sound complex, but using a macro is easy. Having defined the 'DelayMS' macro as above, it can be called from the main loop, as follows:

```
main_loop
    bsf     FLASH            ; turn on LED
    DelayMS .200             ; stay on for 200 ms
    bcf     FLASH            ; turn off LED
    DelayMS .800             ; stay off for 800 ms
    goto    loop             ; repeat forever
```

This 'DelayMS' macro is an example of a *wrapper*, making the 'delay10' subroutine easier to use.

Note that the `pagesel` directives have been included as part of the macro, first to select the correct page for the 'delay10' subroutine, and then to select the current page again after the subroutine call. That makes the macro transparent to use; there is no need for `pagesel` directives before or after calling it.

As a more complex example, consider the debounce code presented in [lesson 5](#):

```
wait_dn clrf    TMR0           ; reset timer
chk_dn  btfsc  GPIO,3        ; check for button press (GP3 low)
        goto   wait_dn       ; continue to reset timer until button down
        movf   TMR0,w        ; has 10ms debounce time elapsed?
        xorlw  .157          ; (157 = 10ms/64us)
        btfss STATUS,Z      ; if not, continue checking button
        goto   chk_dn
```

If you had a number of buttons to debounce in your application, you would want to use code very similar to this, multiple times. But since there is no way of passing a reference to the pin to debounce (such as ‘GPIO,3’) as a parameter to a subroutine, it’s necessary to use a macro to achieve this.

For example, a debounce macro could be defined as follows:

```
; Debounce switch on given input port,pin
; Waits for switch to be 'high' continuously for 10 ms
;
; Uses:      TMR0           Assumes: TMR0 running at 256 us/tick
;
DbnceHi MACRO port,pin
    local    start,wait,DEBOUNCE
    variable DEBOUNCE=.10*.1000/.256 ; debounce count = 10ms/(256us/tick)

    pagesel $           ; select current page for gotos
start   clrf    TMR0     ; button down, so reset timer (counts "up" time)
wait    btfss  port,pin ; wait for switch to go high (=1)
        goto   start
        movf   TMR0,w   ; has switch has been up continuously for
        xorlw  DEBOUNCE ;   debounce time?
        btfss STATUS,Z ; if not, keep checking that it is still up
        goto   wait
    ENDM
```

There are a few things to note about this macro definition, starting with the comments. As with subroutines, you’ll eventually build up a library of useful macros, which you might keep together in an include file, such as ‘stdmacros.inc’ (which you would reference using the #include directive, instead of copying the macros into your code.) When documenting a macro, it’s important to note any resources (such as timers) used by the macro, and any initialisation that has to have been done before the macro is called.

The macro is called ‘DbnceHi’ instead of ‘DbnceUp’ because it’s waiting for a pin to be consistently high. For some switches, that will correspond to “up”, but not in every case. Using terms such as “high” instead of “up” is more general, and thus more reusable.

The ‘local’ directive declares symbols (address labels and variables) which are only used within the macro. If you call a macro more than once, you must declare any address labels within the macro as “local”, or else the assembler will complain that you have used the same label more than once. Declaring macro labels as local also means that you don’t need to worry about whether those labels are used within the main body of code. A good example is ‘start’ in the definition above. There is a good chance that there will be a ‘start’ label in the main program, but that doesn’t matter, as the *scope* of a label declared to be “local” is limited to the macro it is defined in.

The ‘variable’ directive is very similar to the ‘constant’ directive, introduced earlier. The only difference is that the symbol it defines can be updated later. Unlike a constant, the value of a variable can be changed after it has been defined. Other than that, they can be used interchangeably.

In this case, the symbol ‘DEBOUNCE’ is being defined as a variable, but is used as a constant. It is never updated, being used to make it easy to change the debounce period from 10 ms if required, without having to

find the relevant instruction within the body of the macro (and note the way that an arithmetic expression has been used, to make it easy to see how to set the debounce to some other number of milliseconds).

So why define 'DEBOUNCE' as a variable, instead of a constant? If it was defined as a constant, there would potentially be a conflict if there was another constant called 'DEBOUNCE' defined somewhere else in the program. But surely declaring it to be "local" would avoid that problem? Unfortunately, the 'local' directive only applies to labels and variables, not constants. And that's why 'DEBOUNCE' is declared as a "local variable". Its scope is limited to the macro and will not affect anything outside it. You can't do that with constants.

Finally, note that the macro begins with a 'pagesel \$' directive. That is placed there because we cannot assume that the page selection bits are set to the current page when the macro is called. If the current page was not selected, the 'goto' commands within the macro body would fail; they would jump to a different page. That illustrates another difference between macros and subroutines: when a subroutine is called, the page the subroutine is on must have been selected (or else it couldn't have been called successfully), so any 'goto' commands within the subroutine will work. You can't safely make that assumption for macros.

Complete program

The following program demonstrates how this "debounce" macro is used in practice.

It is based on the "toggle an LED" program included in [lesson 5](#), but the press of the pushbutton is not debounced, only the release. It is not normally necessary to debounce both actions – although you may have to think about it a little to see why!

Using the macro doesn't make the code any shorter, but the main loop is much simpler:

```

;*****
;
; Description:      Lesson 6, example 4
;                  Toggles LED when button is pressed
;
; Demonstrates use of macro defining Timer0-based debounce routine
;
;*****
;
; Pin assignments:
;   GP1 = indicator LED
;   GP3 = pushbutton switch (active low)
;
;*****

list      p=12F509
#include   <p12F509.inc>

;***** CONFIGURATION
                ; int reset, no code protect, no watchdog, int RC clock
__CONFIG      _MCLRE_OFF & _CP_OFF & _WDT_OFF & _Intrc_OSC

; pin assignments
constant      nLED=1                ; indicator LED on GP1
#define       BUTTON GPIO,3          ; pushbutton on GP3

;***** MACROS
; Debounce switch on given input port,pin
; Waits for switch to be 'high' continuously for 10 ms
;
; Uses:        TMR0                Assumes: TMR0 running at 256 us/tick
;

```

```

DbnceHi MACRO    port,pin
    local        start,wait,DEBOUNCE
    variable     DEBOUNCE=.10*.1000/.256 ; debounce count = 10ms/(256us/tick)

    pagesel $           ; select current page for gotos
start    clrf    TMR0     ; button down, so reset timer (counts "up" time)
wait     btfss   port,pin ; wait for switch to go high (=1)
         goto    start
         movf    TMR0,w    ; has switch has been up continuously for
         xorlw   DEBOUNCE  ;   debounce time?
         btfss   STATUS,Z  ; if not, keep checking that it is still up
         goto    wait
        ENDM

;***** VARIABLE DEFINITIONS
        UDATA_SHR
sGPIO   res 1           ; shadow copy of GPIO

;***** RC CALIBRATION
RCCAL   CODE    0x3FF   ; processor reset vector
        res 1           ; holds internal RC cal value, as a movlw k

RESET   CODE    0x000   ; effective reset vector
        movwf   OSCCAL  ; apply internal RC factory calibration

;***** MAIN PROGRAM *****
;***** Initialisation
start
    ; configure ports
    clrf    GPIO        ; start with LED off
    clrf    sGPIO       ;   update shadow
    movlw   ~(1<<nLED)  ; configure LED pin (only) as output
    tris    GPIO
    ; configure timer
    movlw   b'11010111' ; configure Timer0:
                ; --0-----   timer mode (T0CS = 0)
                ; ----0----   prescaler assigned to Timer0 (PSA = 0)
                ; -----111   prescale = 256 (PS = 111)
    option ;   -> increment every 256 us

;***** Main loop
main_loop
    ; wait for button press
wait_dn btfsc   BUTTON   ; wait until button low
        goto    wait_dn

    ; toggle LED
    movf    sGPIO,w
    xorlw   1<<nLED      ; toggle shadow register
    movwf   sGPIO
    movwf   GPIO        ; write to port

    ; wait for button release
    DbnceHi BUTTON      ; wait until button high (debounced)

    ; repeat forever
    goto    main_loop

    END

```

Conditional Assembly

We've seen how the processor include files, such as 'p12F509.inc', define a number of symbols that allow you to refer to registers and flags by name, instead of numeric value.

While looking at the 'p12F509.inc' file, you may have noticed these lines:

```
IFNDEF __12F509
    MESSG "Processor-header file mismatch. Verify selected processor."
ENDIF
```

This is an example of *conditional assembly*, where the actions performed by the assembler (outputting messages and generating code) depend on whether specific conditions are met.

When the processor type is specified by the 'list p=' directive, or selected in MPLAB, a symbol specifying the processor is defined; for the PIC12F509, the symbol is '__12F509'. This is useful because the assembler can be made to perform different actions depending on which processor symbol has been defined.

In this case, the idea is to check that the correct processor include file is being used. If you include the file for the wrong processor, you'll almost certainly have problems. This code checks for that.

The 'IFNDEF' directive instructs the assembler to assemble the following block of code if the specified symbol *has not* been defined.

The 'ENDIF' directive marks the end of the block of conditionally-assembled code.

In this case, everything between 'IFNDEF' and 'ENDIF' is assembled if the symbol '__12F509' has not been defined. And that will only be true if a processor other than the PIC12F509 has been selected.

The 'MESSG' directive tells the assembler to print the specified message in the MPLAB output window. This message is only informational; it's useful for providing information about the assembly process or for issuing warnings that do not necessarily mean that assembly has to stop.

So, this code tests that the correct processor has been selected and, if not, warns the user about the mismatch.

Similar to 'IFNDEF', there is also an 'IFDEF' directive which instructs the assembler to assemble a block of code if the specified symbol *has* been defined.

A common use of 'IFDEF' is when debugging, perhaps to disable parts of the program while it is being debugged. Or you might want to use a different processor configuration, say with code protection enabled.

For example:

```
***** CONFIGURATION
#define      DEBUG

IFDEF DEBUG
    __CONFIG    _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC
ELSE
    __CONFIG    _MCLRE_OFF & _CP_ON & _WDT_OFF & _IntRC_OSC
ENDIF
```

If the 'DEBUG' symbol has been defined (it doesn't have to be set equal to anything, just defined), the first __CONFIG directive is assembled, turning off code protection and the watchdog timer.

The 'ELSE' directive marks the beginning of an alternative block of code, to be assembled if the previous conditional block was not selected for assembly.

That is, if the 'DEBUG' symbol has *not* been defined, the second `__CONFIG` directive is assembled, turning on code protection and the watchdog timer.

When you have finished debugging, you can either comment out the '#define DEBUG' directive, or change 'DEBUG' to another symbol, such as 'RELEASE'. The debugging code will now no longer be assembled.

In many cases, simply testing whether a symbol exists is not enough. You may want the assembler to assemble different sections of code and/or issue different messages, depending on the value of a symbol, or of an expression containing perhaps a number of symbols.

As an example, suppose your code is used to support a number of hardware configurations, or revisions. At some point the printed circuit board may have been revised, requiring different pin assignments. In that case, you could use a block of code similar to:

```

constant    REV='A'                ; hardware revision

; pin assignments
IF REV=='A'
    constant    nLED=1              ; pin assignments for REV A:
    ;          indicator LED on GP1
    #define    BUTTON    GPIO,3    ; pushbutton on GP3
ENDIF
IF REV=='B'
    constant    nLED=0              ; pin assignments for REV B:
    ;          indicator LED on GP0
    #define    BUTTON    GPIO,2    ; pushbutton on GP2
ENDIF
IF REV!='A' && REV!='B'
    ERROR "Revision must be 'A' or 'B'"
ENDIF

```

This code allows for two hardware revisions, selected by setting the constant 'REV' equal to 'A' or 'B'.

It's easy to try this out with your Gooligum baseline training board: close jumpers JP7 and JP3 to enable pull-up resistors on GP2 and GP3, and jumpers JP11 and JP12 to enable LEDs on GP0 and GP1.

The '`IF expr`' directive instructs the assembler to assemble the following block of code if the expression `expr` is true. Normally a *logical expression* (such as a test for equality) is used with the 'IF' directive, but arithmetic expressions can also be used, in which case an expression that evaluates to zero is considered to be logically false, while any non-zero value is considered to be logically true.

MPASM supports the following logical operators:

not (logical compliment)	!
greater than or equal to	>=
greater than	>
less than	<
less than or equal to	<=
equal to	==
not equal to	!=
logical AND	&&
logical OR	

Precedence is in the order shown.

As usual, parentheses can be used to change the order of precedence: '(' and ')'.

Note that the test for equality is two equals signs; '==', not '='.

In the code above, setting 'REV' to 'A' means that the first pair of #define directives will be executed, while setting 'REV' to 'B' executes the second pair.

But what if 'REV' was set to something other than 'A' or 'B'? Then neither set of pin assignments would be selected and the symbols 'nLED' and 'BUTTON' would be left undefined. The rest of the code would not assemble correctly, so it is best to check for that error condition.

This error condition can be tested for, using the logical expression:

```
REV!='A' && REV!='B'
```

Incidentally, this can be rewritten equivalently¹ as:

```
!(REV=='A' || REV=='B')
```

You can of course use whichever form seems clearest to you.

The 'ERROR' directive does essentially the same thing as 'MESSG', but instead of printing the specified message and continuing, 'ERROR' will make the assembly process fail.

The 'IF' directive is also very useful for checking that macros have been called correctly, particularly for macros which may be reused in other programs.

For example, consider the delay macro defined earlier:

```
DelayMS MACRO    ms                ; delay time in ms
    movlw    ms/.10                ; divide by 10 to pass to delay10 routine
    pagesel  delay10
    call     delay10
    pagesel  $
ENDM
```

The maximum delay allowed is 2.55 s, because all the registers, including W, are 8-bit and so can only hold numbers up to 255. If you try calling 'DelayMS' with an argument greater than 2550, the assembler will warn you about "Argument out of range", but it will carry on anyway, using the least significant 8 bits of 'ms/.10'. That's not a desirable behaviour. It would be better if the assembler reported an error and halted, if the macro is called with an argument that is out of range.

That can be done as follows:

```
DelayMS MACRO    ms                ; delay time in ms
    IF ms>2550
        ERROR "Maximum delay time is 2550 ms"
    ENDIF
    movlw    ms/.10                ; divide by 10 to pass to delay10 routine
    pagesel  delay10
    call     delay10
    pagesel  $
ENDM
```

By testing that parameters are within allowed ranges like this, you can make your code more robust.

Conclusion

MPASM offers many more advanced facilities that can make your life as a PIC assembler programmer easier, but that's enough for now.

¹ This equivalence is known as De Morgan's theorem.

The features we've seen in this lesson will help make your code easier to understand and maintain, and make you more productive, by:

- using arithmetic expressions to make it clear how numeric constants are derived
- using constants and text substitution, so make your code more readable and so that future configuration changes can be made in a single place
- using symbols and bitwise operators instead of cryptic binary constants
- creating macros to make your code shorter, more readable, and easier to re-use

Other MPASM directives will be introduced in future lessons, as appropriate.

The [next lesson](#) covers the 12F509's sleep mode, watchdog timer, and clock (oscillator) options.

Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 7: Sleep Mode, the Watchdog Timer and Clock Options

We've now covered, at least at an introductory level, the major features of the PIC10F200, PIC12F508 and PIC12F509 (admittedly, some of the simplest of the “modern” PICs), including digital input, output, and using the Timer0 module as either a timer or counter.

That's enough to build a surprising number of applications, but these MCUs have a few other features which can be quite useful. These are covered in chapter 7 of the PIC12F508/509/16F505 data sheet, titled “Special Features of the CPU”. Although you should refer to the latest data sheet for the full details, this lesson will introduce the following “special” (and very useful) features:

- Sleep mode (power down)
- Wake-up on change (power up)
- The watchdog timer
- Oscillator (clock) configurations

Sleep Mode

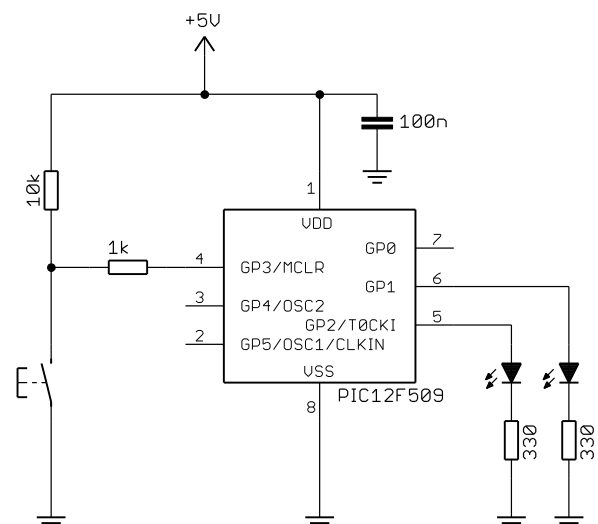
The material covered so far in these tutorials should allow you to design a simple project such as the Gooligum Electronics “[Toy Traffic Lights](#)” kit: lighting LEDs, responding to and debouncing buttons and switches, and timing. But there's one thing the Toy Traffic Lights project does, that hasn't been covered yet; it turns itself “off” (saving power), and comes back “on” at the touch of a button. There is no on/off switch; the circuit is always powered, and yet the batteries are able to last for years.

That is done by putting the PIC into the power-saving standby, or *sleep* mode.

To demonstrate sleep mode, we'll use the circuit from [lesson 5](#), as shown on the right.

If you have the [Gooligum baseline training board](#), connect jumpers JP3, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP1 and GP2. Or, if you are using Microchip's Low Pin Count Demo Board, you will need to connect LEDs to GP1 and GP2, as described in [lesson 1](#).

To demonstrate to yourself that power consumption really is reduced when the PIC enters sleep mode, you would have to use an external power supply, instead of using your PICkit 2 or PICkit 3 to power the circuit. You can then place a multimeter inline with the power supply, to measure the supply current.



The instruction for placing the PIC into standby mode is ‘sleep’ – “enter **sleep** mode”.

To illustrate the use of the sleep instruction, consider the following fragment of code. It turns on the LED on GP1, waits for the button to be pressed, and then enters sleep mode:

```

        movlw   b'111101'      ; configure GP1 (only) as an output
        tris   GPIO

        bsf    GPIO,1         ; turn on LED

wait_lo btfsc   GPIO,3         ; wait for button press (low)
        goto   wait_lo

        sleep                  ; enter sleep mode

        goto   $              ; (this instruction should never run)

```

Note that the final ‘goto \$’ instruction (an endless loop) will never be executed, because ‘sleep’ will halt the processor; any instructions after ‘sleep’ will never be reached.

When you run this program, the LED will turn on and then, when you press the button, nothing will appear to happen! The LED stays on. Shouldn’t it turn off? What’s going on?

The current supplied from a 5 V supply, before pressing the button, with the LED on, was measured to be 10.83 mA. After pressing the button, the measured current dropped to 10.47 mA, a fall of only 0.36 mA.

This happens because, when the PIC goes into standby mode, the PIC stops executing instructions, saving some power (360 μ A in this case), but the I/O ports remain in the state they were in, before the ‘sleep’ instruction was executed.

Note: For low power consumption in standby mode, the I/O ports must be configured to stop sourcing or sinking current, before entering SLEEP mode.

In this case, the fix is simple – turn off the LED before entering sleep mode, as follows:

```

        movlw   b'111101'      ; configure GP1 (only) as an output
        tris   GPIO

        bsf    GPIO,1         ; turn on LED

wait_lo btfsc   GPIO,3         ; wait for button press (low)
        goto   wait_lo

        bcf    GPIO,1         ; turn off LED

        sleep                  ; enter sleep mode

```

When this program is run, the LED will turn off when the button is pressed.

The current measured in the prototype with the PIC in standby and the LED off was less than 0.1 μ A – too low to register on the multimeter used! That was with the unused pins tied to VDD or VSS (whichever is most convenient on the circuit board), as floating CMOS inputs will draw unnecessary current.

Note: To minimise power in standby mode, configure all unused pins as inputs, and tie them VDD or VSS through resistors (do not connect them directly to VDD or VSS, as the PIC may be damaged if these pins are inadvertently configured as outputs).

For clarity, tying the unused inputs to VDD or VSS was not shown in the circuit diagram above. If you have the Gooligum training board, you can use the supplied 22 kΩ resistors to tie GP0, GP4 and GP5 to ground, via the expansion header and breadboard.

Wake-up from sleep

Most baseline PICs include a facility for coming out of standby mode when an input changes, called *wake-up on change*. This is used, for example, in the “[Toy Traffic Lights](#)” project to power on the device when the button is pressed.

Wake-up on change is available on the GP0, GP1 and GP3 pins on the PIC12F509 (these are the same pins that internal pull-ups are available for). Note that on the baseline PICs, this is all or nothing; either all of the available pins are enabled for wake-up on change, or none of them are.

On the PIC12F509, wake-up on change is controlled by the $\overline{\text{GPWU}}$ bit in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION	$\overline{\text{GPWU}}$	$\overline{\text{GPPU}}$	T0CS	T0SE	PSA	PS2	PS1	PS0

By default (after a power-on or reset), $\overline{\text{GPWU}} = 1$ and wake-up on change is disabled.

To enable internal wake-up on change, clear $\overline{\text{GPWU}}$.

Assuming no other options are being set (leaving all the other bits at the default value of ‘1’), wake-up on change is enabled by:

```
movlw    b'01111111'    ; enable wake-up on change
          ; 0-----    (/GPWU = 0)
option
```

If wake-up on change is enabled, the PIC will be reset if, in sleep mode, the value at any of the “wake-up on change” pins becomes different to the last time those pins were read, prior to entering sleep.

Note: You should read the input pins configured for wake-up on change just prior to entering sleep mode. Otherwise, if the value at a pin had changed since the last time it was read, a “wake up on change” reset will occur immediately upon entering sleep mode, as the input value would be seen to be different from that last read.

It is also important to ensure that any input which will be used to trigger a wake-up is stable before entering sleep mode. Consider what would happen if wake-up on change was enabled in the program above. As soon as the button is pressed, the LED will turn off and the PIC will enter standby mode, as intended. But on the first switch bounce, the input would be seen to have changed, and the PIC would be reset.

Even if the circuit included hardware debouncing, there’s still a problem: the LED will go off and the PIC will enter standby as soon as the button is pressed, but when the button is subsequently released, it will be seen as a change, and the PIC will reset and the LED will come back on! To successfully use the pushbutton to turn the circuit (PIC and LED) “off”, it is necessary to wait for the button to go high and remain stable (debounced) before entering sleep mode.

There’s another problem: when the button is pressed while the PIC is in sleep mode, the PIC will reset, and the LED will light. That’s what we want. The problem is that PICs are fast, and human fingers are slow – the button will still be down when the program first checks for “button down” and the LED will immediately turn off again. To avoid this, we must wait for the button to be in a stable “up” state before checking that it is “down”, in case the program is starting following a button press.

So the necessary sequence is:

```
turn on LED
wait for stable button high
wait for button low
turn off LED
wait for stable button high
sleep
```

The following code, which makes use of the debounce macro defined in [lesson 6](#), implements this:

```
***** Initialisation
    ; configure port
    movlw  ~(1<<nLED)      ; configure LED pin (only) as an output
    tris   GPIO
    ; configure wake-on-change and Timer0
    movlw  b'01000111'    ; configure wake-up on change and Timer0:
    ; 0-----          enable wake-up on change (/GPWU = 0)
    ; --0-----        timer mode (T0CS = 0)
    ; ----0---         prescaler assigned to Timer0 (PSA = 0)
    ; -----111       prescale = 256 (PS = 111)
    option                ; -> increment every 256 us

***** Main code
    ; turn on LED
    bsf    LED

    ; wait for stable button high (in case it is still bouncing)
    DbnceHi BUTTON

    ; wait for button press
wait_lo btfsc  BUTTON      ; wait until button low
        goto  wait_lo

    ; go into standby (low power) mode
    bcf    LED            ; turn off LED

    DbnceHi BUTTON      ; wait for stable button release

    sleep                ; enter sleep mode
```

(the labels 'LED' and 'BUTTON' are defined earlier in the program; see the complete listing below)

This code does essentially the same thing as the “toggle a LED” programs developed in [lesson 4](#), except that in this case, when the LED is off, the PIC is drawing negligible power.

*Note: On baseline PICs, wake-up on pin change causes a processor reset; instruction execution recommences from the reset vector, as it does following all types of reset, including power-on. Execution does **not** resume at the instruction following “sleep”.*

Since the same start-up instructions are executed, whether the PIC has been powered on for the first time, or was reset by a wake-up from sleep, how is it possible to tell whether a wake-up on change has occurred?

Of course, that’s not necessarily important. The program above debounces the pushbutton when it first starts, just in case it had restarted because of a wake-up from sleep. If the PIC had just been powered on, there would be no need to do this debouncing, but it doesn’t hurt to do it anyway – if the button is already up, then the debounce routine only introduces a 10 ms delay.

But sometimes you would like your program to behave differently, depending on why it was (re)started.

You can do that by testing the GPWUF flag bit in the STATUS register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	GPWUF	-	PA0	\overline{TO}	\overline{PD}	Z	DC	C

GPWUF is set to '1' by a wake-up on change, and is cleared by all other resets. So if GPWUF has been set, it means that a wake-on-change reset has occurred.

Complete program

To demonstrate how the GPWUF flag can be tested, to differentiate between wake-up on change and power-on resets, the following program, based on the code above, lights the LED on GP2 following a wake-up on change reset, but not when the PIC is first powered on. And since the wake-up on change condition is being tested anyway, the initial button debounce is only performed if a wake-up on change has occurred. (Note that the debounce macro is defined in an include file.)

```

;*****
;
; Description: Lesson 7, example 2b
;
; Demonstrates differentiation between wake up on change
; and POR reset
;
; Turn on LED after each reset
; Turn on WAKE LED only if reset was due to wake on change
; then wait for button press, turn off LEDs, debounce, then sleep
;
;*****
; Pin assignments:
; GP1 = on/off indicator LED
; GP2 = wake-on-change indicator LED
; GP3 = pushbutton switch (active low)
;
;*****

list      p=12F509
#include  <p12F509.inc>

#include  <stdmacros-base.inc> ; DbcneHi - debounce switch, wait for high
                                ; (requires TMR0 running at 256 us/tick)
radix    dec

;***** CONFIGURATION
                ; int reset, no code protect, no watchdog, int RC clock
__CONFIG      _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC

; pin assignments
#define LED      GPIO,1      ; on/off indicator LED on GP1
constant nLED=1             ; (port bit 1)
#define WAKE     GPIO,2      ; wake on change indicator LED on GP2
constant nWAKE=2           ; (port bit 2)
#define BUTTON   GPIO,3      ; pushbutton on GP3 (active low)

;***** RC CALIBRATION
RCCAL CODE 0x3FF           ; processor reset vector
res 1                     ; holds internal RC cal value, as a movlw k

```

```

;***** RESET VECTOR *****
RESET   CODE      0x000          ; effective reset vector
        movwf     OSCCAL         ; apply internal RC factory calibration

;***** MAIN PROGRAM *****

;***** Initialisation
start
        ; configure port
        clrf      GPIO          ; start with all LEDs off
        movlw    ~(1<<nLED|1<<nWAKE) ; configure LED pins as outputs
        tris     GPIO
        ; configure wake-on-change and Timer0
        movlw    b'01000111'    ; configure wake-up on change and Timer0:
        ; 0-----          enable wake-up on change (/GPWU = 0)
        ; --0-----        timer mode (T0CS = 0)
        ; ----0----        prescaler assigned to Timer0 (PSA = 0)
        ; -----111       prescale = 256 (PS = 111)
        option   ; -> increment every 256 us

;***** Main code
        ; turn on LED
        bsf      LED

        ; test for wake-on-change reset
        btfss   STATUS,GPWUF    ; if wake-up on change has occurred,
        goto    wait_lo
        bsf     WAKE            ; turn on wake-up indicator
        DbncHi  BUTTON         ; wait for button to stop bouncing

        ; wait for button press
wait_lo btfsc   BUTTON          ; wait until button low
        goto    wait_lo

        ; go into standby (low power) mode
        clrf    GPIO          ; turn off LEDs

        DbncHi  BUTTON         ; wait for stable button release

        sleep   ; enter sleep mode

        END

```

Watchdog Timer

In the real world, computer programs sometimes “crash”; they will stop responding to input, stuck in a continuous loop they can’t get out of, and the only way out is to reset the processor (e.g. Ctrl-Alt-Del on Windows PCs – and even that sometimes won’t work, and you need to power cycle a PC to bring it back). Microcontrollers are not immune to this. Their programs can become stuck because some unforeseen sequence of inputs has occurred, or perhaps because an expected input signal never arrives. Or, in the electrically noisy industrial environment in which microcontrollers are often operating, power glitches and EMI on signal lines can create an unstable environment, perhaps leading to a crash.

Crashes present a special problem for equipment which is intended to be reliable, operating autonomously, in environments where user intervention isn’t an option.

One of the major functions of a *watchdog timer* is to automatically reset the microcontroller in the event of a crash. It is simply a free-running timer (running independently of any other processor function, including sleep) which, if allowed to overflow, will reset the PIC. In normal operation, an instruction which clears the watchdog timer is regularly executed – often enough to prevent the timer ever overflowing. This instruction is often placed in the “main loop” of a program, where it would normally be expected to be executed often enough to prevent watchdog timer overflows. If the program crashes, the main loop presumably won’t complete; the watchdog timer won’t be cleared, and the PIC will be reset. Hopefully, when the PIC restarts, whatever condition led to the crash will have gone away, and the PIC will resume normal operation.

The instruction for clearing the watchdog timer is ‘`clrwdt`’ – “clear watchdog timer”.

The watchdog timer has a nominal time-out period of 18 ms. If that’s not long enough, it can be extended by using the prescaler.

As we saw in [lesson 5](#), the prescaler is configured using a number of bits in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION	GPWU	GPPU	T0CS	T0SE	PSA	PS2	PS1	PS0

To assign the prescaler to the watchdog timer, set the PSA bit to ‘1’.

Note: The baseline PICs include a single prescaler, which can be used with either the Timer0 module or the Watchdog Timer, but not both.

If the prescaler is assigned to the Watchdog Timer, it cannot be used with Timer0.

When assigned to the watchdog timer, the prescale ratio is set by the PS<2:0> bits, as shown in the table on the right.

PS<2:0> bit value	WDT prescale ratio
000	1 : 1
001	1 : 2
010	1 : 4
011	1 : 8
100	1 : 16
101	1 : 32
110	1 : 64
111	1 : 128

Note that the prescale ratios are one half of those that apply when the prescaler is assigned to Timer0.

For example, if PSA = 1 (assigning the prescaler to the watchdog timer) and PS<2:0> = ‘011’ (selecting a ratio of 1:8), the watchdog time-out period will be 8 × 18 ms = 144 ms.

With the maximum prescale ratio, the watchdog time-out period is 128 × 18 ms = 2.3 seconds.

The watchdog timer is controlled by the WDTE bit in the configuration word:

Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	-	-	-	-	MCLRE	CP	WDTE	FOSC1	FOSC0

Setting WDTE to ‘1’ enables the watchdog timer.

To set WDTE, use the symbol ‘`_WDT_ON`’ instead of ‘`_WDT_OFF`’ in the `__CONFIG` directive.

Since the configuration word cannot be accessed by programs running on the PIC (it can only be written to when the PIC is being programmed), **the watchdog timer cannot be enabled or disabled at runtime**. It can only be configured to be ‘on’ or ‘off’ when the PIC is programmed.

Watchdog Timer example

To demonstrate how the watchdog timer allows the PIC to recover from a crash, we’ll use a simple program which turns on an LED for 1.0 s, turns it off again, and then enters an endless loop (simulating a crash).

If the watchdog timer is disabled, the loop will never exit and the LED will remain off. But if the watchdog timer is enabled, with a period of 2.3 s, the program should restart itself after 2.3 s, and the LED will flash: on for 1.0 sec and off for 1.3 s (approximately).

To make it easy to select between configurations with the watchdog timer on or off, you can use a construct such as:

```
#define WATCHDOG ; define to enable watchdog timer

IFDEF WATCHDOG
    __CONFIG _MCLRE_ON & _CP_OFF & _WDT_ON & _IntRC_OSC
ELSE
    __CONFIG _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC
ENDIF
```

Note that these `__CONFIG` directives enable external reset (`_MCLRE_ON`), allowing the pushbutton switch connected to pin 4 to reset the PIC. That’s useful because, with this program going into an endless loop, having to power cycle the PIC to restart it would be annoying; pressing the button is much more convenient.

The code to flash the LED once and then enter an endless loop is simple, making use of the ‘DelayMS’ macro introduced in [lesson 6](#):

```
***** Initialisation
start
    ; configure port
    movlw ~ (1<<nLED) ; configure LED pin (only) as an output
    tris GPIO
    ; configure watchdog timer
    movlw 1<<PSA | b'111' ; prescaler assigned to WDT (PSA = 1)
    ; prescale = 128 (PS = 111)
    option ; -> WDT period = 2.3 s

***** Main code
    bsf LED ; turn on LED
    DelayMS 1000 ; delay 1 sec
    bcf LED ; turn off LED

    goto $ ; wait forever
```

If you build and run this with `#define WATCHDOG` commented out (place a ‘;’ in front of it), the LED will light once, and then remain off. But if you define ‘WATCHDOG’, the LED will continue to flash.

As mentioned in the discussion of “wake-up on change”, sometimes you’d like your program to behave differently, depending on why it was restarted. In particular, if, in normal operation, a watchdog timer reset should never occur, you may wish to turn on an alarm indicator if a watchdog timer reset has happened, to show that an unexpected problem has occurred.

Watchdog timer resets are indicated by the $\overline{\text{TO}}$ bit in the STATUS register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	GPWUF	-	PA0	$\overline{\text{TO}}$	$\overline{\text{PD}}$	Z	DC	C

The $\overline{\text{TO}}$ bit is cleared to '0' by a reset caused by watchdog timer, and is set to '1' by power-on reset, entering sleep mode, or by execution of the 'clrwdt' instruction.

If $\overline{\text{TO}}$ has been cleared, it means that a watchdog timer reset has occurred.

To demonstrate how the $\overline{\text{TO}}$ flag is used, the code above can be modified, to light the LED if a watchdog timer reset has occurred, but not when the PIC is first powered on, as follows:

```

;***** Initialisation
start
    ; configure port
    clrf    GPIO                ; start with all LEDs off
    movlw  ~(1<<nLED|1<<nWDT)    ; configure LED pins as outputs
    tris   GPIO
    ; configure watchdog timer
    movlw  1<<PSA | b'111'      ; prescaler assigned to WDT (PSA = 1)
                                ; prescale = 128 (PS = 111)
    option                                ; -> WDT period = 2.3 s

;***** Main code
    ; test for WDT-timeout reset
    btfss  STATUS,NOT_TO        ; if WDT timeout has occurred,
    bsf    WDT                  ; turn on "error" LED

    ; flash LED
    bsf    LED                  ; turn on "flash" LED
    DelayMS 1000                ; delay 1 sec
    bcf    LED                  ; turn off "flash" LED

    ; wait forever
    goto  $

```

Of course, you will normally want to avoid watchdog timer resets.

As discussed earlier, to prevent the watchdog timer timing out, simply place a 'clrwdt' instruction within the main loop, with the watchdog timer period set to be longer than it should ever take to complete the loop.

To demonstrate that the 'clrwdt' instruction really does stop the watchdog expiring (if executed often enough), simply include it in the endless loop at the end of the code:

```

loop    clrwdt                ; clear watchdog timer
        goto    loop          ; repeat forever

```

If you replace the 'goto \$' line with this "clear watchdog timer" loop, you will find that, after flashing once, the LED remains off – regardless of the watchdog timer setting.

Periodic wake from sleep

The watchdog timer can also be used to wake the PIC from sleep mode.

This is useful in situations where inputs do not need to be responded to instantly, but can be checked periodically. To minimise power drain, the PIC can sleep most of the time, waking up every so often (say, once per second), checking inputs and, if there is nothing to do, going back to sleep.

Note that a periodic wake-up can be combined with wake-up on pin change; you may for example wish to periodically log the value of a sensor, but also respond immediately to button presses.

Setting up a periodic wake-up is easy: simply configure the watchdog timer for the desired wake-up period, perform the “main code” tasks (testing and responding to inputs), then enter sleep mode. When the watchdog timer period has elapsed, the PIC will wake up, perform the main tasks, and then go to sleep again.

To illustrate this process, we can simply replace the endless loop with a ‘sleep’ instruction:

```

;***** Initialisation
start
    ; configure port
    movlw    ~(1<<nLED)          ; configure LED pin (only) as an output
    tris    GPIO
    ; configure watchdog timer
    movlw    1<<PSA | b'111'     ; prescaler assigned to WDT (PSA = 1)
                                ; prescale = 128 (PS = 111)
    option                                     ; -> WDT period = 2.3 s

;***** Main code
    bsf     LED                  ; turn on LED
    DelayMS 1000                ; delay 1 sec
    bcf     LED                  ; turn off LED

    sleep                       ; enter sleep mode

```

You’ll find that the LED is on for 1 s, and then off for around 2 s. That is because the watchdog timer is cleared automatically when the PIC enters sleep mode.

Oscillator (Clock) Options

Every example in these lessons, until now, has used the 4 MHz internal RC oscillator as the PIC’s clock source. It’s usually a very good option – simple to use, needing no external components, using none of the PIC pins, and reasonably accurate.

However, there are situations where it is more appropriate to use some external clock circuitry.

Reasons to use external clock circuitry include:

- *Greater accuracy and stability.*
A crystal or ceramic resonator is significantly more accurate than the internal RC oscillator, with less frequency drift due to temperature and voltage variations.
- *Generating a specific frequency.*
For example, as we saw in lesson 5, the signal from a 32.768 kHz crystal can be readily divided down to 1Hz. Or, to produce accurate timing for RS-232 serial data transfers, a crystal frequency such as 1.843200 MHz can be used, since it is an exact multiple of common baud rates, such as 38400 or 9600 ($1843200 = 48 \times 38400 = 192 \times 9600$).
- *Synchronising with other components.*
Sometimes it simplifies design if a number of microcontrollers (or other chips) are clocked from a common source, so that their outputs change synchronously – although you need to be careful; clock signals which are subject to varying delays in a circuit will not be synchronised in practice (a phenomenon known as *clock skew*), leading to unpredictable results.
- *Lower power consumption.*
At a given supply voltage, PICs draw less current when they are clocked at a lower speed. For example, the PIC12F508/509 data sheet states (parameter D010) that, with $V_{DD} = 2.0$ V, supply current is typically 170 μ A for a clock speed of 4 MHz, but only 15 μ A at 32 kHz. Power consumption can be minimised by running the PIC at the slowest practical clock speed. And for many applications, very little speed is needed.

PICs support a number of clock, or oscillator, configurations, allowing, through appropriate oscillator selection, any of these goals to be met (but not necessarily all at once – low power consumption and high frequencies don't mix!)

The oscillator configuration is selected by the FOSC bits in the configuration word:

Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	-	-	-	-	MCLRE	\overline{CP}	WDTE	FOSC1	FOSC0

The PIC12F508 and 509 have two FOSC bits, allowing selection of one of four oscillator configurations, as in the table on the right.

FOSC<1:0>	Oscillator configuration
00	LP oscillator
01	XT oscillator
10	Internal RC oscillator
11	External RC oscillator

The internal RC oscillator is the one we have been using so far, providing a nominal 4 MHz internal clock source, and has already been discussed.

The other oscillator options are described in detail in the PIC12F508/509 data sheet, as well as in a number of application notes available on the Microchip web site, www.microchip.com.

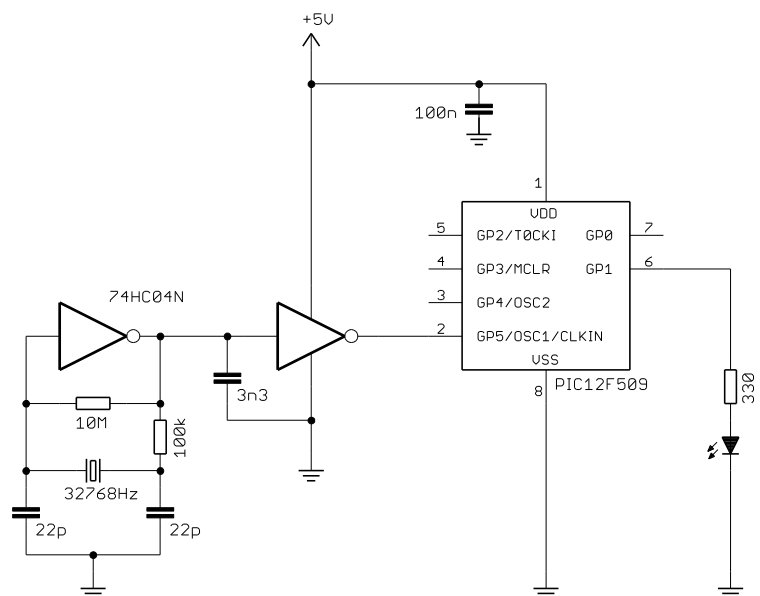
Instead of needlessly repeating all that material here, the following sections outline some of the most common oscillator configurations.

External clock input

An external oscillator can be used to clock the PIC.

As discussed above, this is sometimes done to synchronise various parts of a circuit is to the same clock signal. Or, you may choose to use an existing external clock signal simply because it is available and is more accurate and stable than the PIC's internal RC oscillator – assuming you can afford the loss of two of the PIC's I/O pins.

[Lesson 5](#) included the design for a 32.768 kHz crystal oscillator, shown in the circuit on the right. We can use it to demonstrate how to use an external clock signal.



To use an external oscillator with the PIC12F509, the PIC has to be configured in either 'LP' or 'XT' oscillator mode. You should use 'LP' for frequencies below around 200 kHz, and 'XT' for higher frequencies.

The clock signal is connected to the CLKIN input: pin 2 on a PIC12F509.

To implement this circuit using the [Gooligum baseline training board](#), place a shunt in position 4 ("EC") of jumper block JP20, connecting the 32.768 kHz signal to CLKIN, and in JP12 to enable the LED on GP1.

When using an external clock signal in the ‘LP’ and ‘XT’ oscillator modes, the OSC2 pin (pin 3 on a PIC12F509) is unused; it is left disconnected and the associated I/O pin (GP4) is not available for use.

Many PICs, such as the 16F506, offer an ‘EC’ (external clock) oscillator mode, which leaves the OSC2 pin available for I/O, as we’ll see in the [next lesson](#). But that’s not an option on the 12F509.

To illustrate the operation of this circuit, we can modify the crystal-driven LED flasher program developed in [lesson 5](#). In that program, the external 32.768 kHz signal was used to drive the Timer0 counter.

Now, however, the 32.768 kHz signal is driving the processor clock, giving an instruction clock rate of 8192 Hz. If Timer0 is configured in timer mode with a 1:32 prescale ratio, TMR0<7> will be cycling at exactly 1 Hz (since $8192 = 32 \times 256$) – as is assumed in the main body of the program from [lesson 5](#).

So, to adapt that program for this circuit, all we need to do is change the configuration statement from:

```
__CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC
```

to:

```
__CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _LP_OSC
```

(The `_XT_OSC` option should be used instead of `_LP_OSC` for higher clock frequencies)

and also to change the timer initialisation code from:

```
movlw    b'11110110'    ; configure Timer0:
          ; --1-----    counter mode (T0CS = 1)
          ; ----0---    prescaler assigned to Timer0 (PSA = 0)
          ; -----110    prescale = 128 (PS = 110)
option   ; -> increment at 256 Hz with 32.768 kHz input
```

to:

```
movlw    b'11010100'    ; configure Timer0:
          ; --0-----    timer mode (T0CS = 0)
          ; ----0---    prescaler assigned to Timer0 (PSA = 0)
          ; -----100    prescale = 32 (PS = 100)
option   ; -> increment at 256 Hz with 32.768 kHz clock
```

The LED on GP1 should then flash at almost exactly 1 Hz – to within the accuracy of the crystal oscillator.

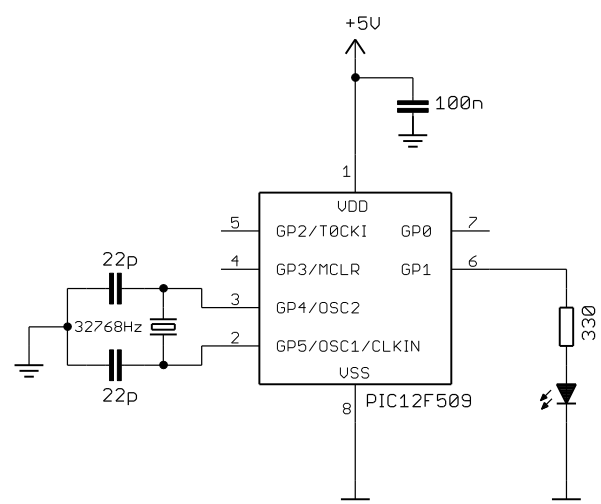
Crystals and ceramic resonators

Generally, there is no need to build your own crystal oscillator; PICs include an oscillator circuit designed to drive crystals directly.

A parallel (not serial) cut crystal, or a ceramic resonator, is placed between the OSC1 and OSC2 pins, which are grounded via loading capacitors, as shown in the circuit diagram on the right.

You should consult the crystal or resonator manufacturer’s data when selecting load capacitors; those shown here are appropriate for a crystal designed for a load capacitance of 12.5 pF.

For some crystals it may be necessary to reduce the drive current by placing a resistor between OSC2 and the crystal, but in most cases it is not needed, and the circuit shown here can be used.



If you are using the [Gooligum baseline training board](#), place shunts in position 2 (“32kHz”) of JP20¹ and position 2 of JP21 (“32kHz”), connecting the 32.768 kHz crystal between OSC1 and OSC2, and in JP12 to enable the LED on GP1.

The PIC12F509 provides two crystal oscillator modes: ‘XT’ and ‘LP’.

They differ in the gain and frequency response of the drive circuitry.

‘XT’ (“crystal”) is the mode used most commonly for crystal or ceramic resonators operating between 200 kHz and 4 MHz.

Lower frequencies generally require lower gain. The ‘LP’ (“low power”) mode uses less power and is designed to drive common 32.786 kHz “watch” crystals, as used in the external clock circuit above, although it can also be used with other low-frequency crystals or resonators.

The circuit as shown here can be used to operate the PIC12F509 at 32.768 kHz, giving low power consumption and an 8192 Hz instruction clock rate, which, as in the external clock example, is easily divided to create an accurate 1 Hz signal.

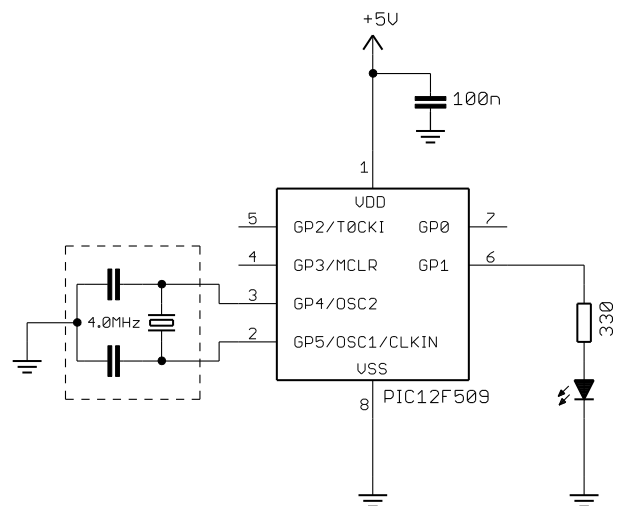
To flash the LED at 1 Hz, the program is exactly the same as for the external clock, including the configuration directive, which **must** include the `_LP_OSC` option:

```
__CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _LP_OSC
```

A convenient option, when you want greater accuracy and stability than the internal RC oscillator can provide, but do not need as much as that offered by a crystal, is to use a ceramic resonator.

These are available in convenient 3-terminal packages which include appropriate loading capacitors, as shown in the circuit diagram on the right. The resonator package incorporates the components within the dashed lines.

If you have the [Gooligum baseline training board](#), move the shunts to position 3 (“4MHz”) of JP20 and position 1 of JP21 (“4MHz”), connecting the 4.0 MHz resonator between OSC1 and OSC2, and leave JP12 closed, to enable the LED on GP1.



To test this circuit, we can use the “flash an LED” program developed in [lesson 2](#); the only change needed is to replace the `_IntRC_OSC` configuration option with `_XT_OSC`, to select crystal oscillator mode.

¹ You will find, with the Gooligum training board, that the LED in the external clock and 32.768 kHz crystal examples will flash, even with no shunt installed in JP20! This is because, when configured in `_LP_OSC` mode, the OSC1 input is very sensitive, and picks up crosstalk from the 32.768 kHz signal on the board. If you want to prevent this effect, you can dampen the 32.768 kHz signal by loading it with a 100 Ω resistor, placed between the 32.768 kHz signal (pin 1 of the expansion header) and ground, via the breadboard. The external clock example will still work with this resistor in place, but only with a shunt in the “EC” position of JP20 – as it should. And the 32.768 kHz crystal example will also then only work with shunts in the “32kHz” positions of JP20 and JP21 – as we’d expect.

However, in [lesson 2](#) we concluded that, since the internal RC oscillator is only accurate to within 1% or so, there was no reason to strive for precise loop timing; a delay of 499.958 ms was considered close enough to the desired 500 ms. Although a ceramic resonator isn't really much more accurate (typically 0.5%), as an exercise we might as well try to make the loop timing as precise as possible – good practice, in case one day you use a crystal oscillator with 50 ppm accuracy!

Therefore in the following program an additional short loop and some `nop` instructions have been added to pad out the total loop time to exactly 500,000 instruction cycles, which will be as close to 500 ms as the accuracy of the resonator or crystal oscillator allows.

Complete program

Here is the complete program, including more precise delay loops:

```

;*****
;
; Description: Lesson 7, example 4c
;
; Demonstrates PIC oscillator, using 4 MHz crystal (or resonator)
;
; LED on GP1 flashes at 1 Hz (50% duty cycle),
; with timing derived from 1 MHz instruction clock
;
;*****
;
; Pin assignments:
; GP1 = flashing LED
; OSC1, OSC2 = 4.00 MHz crystal (or resonator)
;
;*****

list p=12F509
#include <p12F509.inc>

radix dec

;***** CONFIGURATION
; ext reset, no code protect, no watchdog, XT crystal
__CONFIG _MCLRE_ON & _CP_OFF & _WDT_OFF & _XT_OSC

; pin assignments
constant nLED=1 ; flashing LED on GP1

;***** VARIABLE DEFINITIONS
UDATA_SHR
sGPIO res 1 ; shadow copy of GPIO
dc1 res 1 ; delay loop counters
dc2 res 1

;***** RC CALIBRATION
RCCAL CODE 0x3FF ; processor reset vector
res 1 ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET CODE 0x000 ; effective reset vector
movwf OSCCAL ; apply internal RC factory calibration

```

```

;***** MAIN PROGRAM *****
;***** Initialisation
start
    ; configure port
    clrf    GPIO           ; start with GPIO clear (LED off)
    clrf    sGPIO         ; update shadow register
    movlw   ~(1<<nLED)    ; configure LED pin (only) as an output
    tris    GPIO

;***** Main loop
main_loop
    ; toggle LED
    movf    sGPIO,w
    xorlw   1<<nLED        ; flip LED pin bit (shadow)
    movwf   sGPIO
    movwf   GPIO         ; write to GPIO

    ; delay 500 ms
    movlw   .244         ; outer loop: 244 x (1023 + 1023 + 3) + 2
                        ; = 499,958 cycles
    movwf   dc2
    clrf    dc1          ; inner loop: 256 x 4 - 1
dly1      nop           ; inner loop 1 = 1023 cycles
    decfsz  dc1,f
    goto    dly1
dly2      nop           ; inner loop 2 = 1023 cycles
    decfsz  dc1,f
    goto    dly2
    decfsz  dc2,f
    goto    dly1
    movlw   .11          ; delay another 11 x 3 - 1 + 2 = 34 cycles
    movwf   dc2         ; -> delay so far = 499,958 + 34
dly3      decfsz  dc2,f  ; = 499,992 cycles
    goto    dly3
    nop           ; main loop overhead = 6 cycles, so add 2 nops
    nop           ; -> loop time = 499,992 + 6 + 2 = 500,000 cycles

    ; repeat forever
    goto    main_loop

END

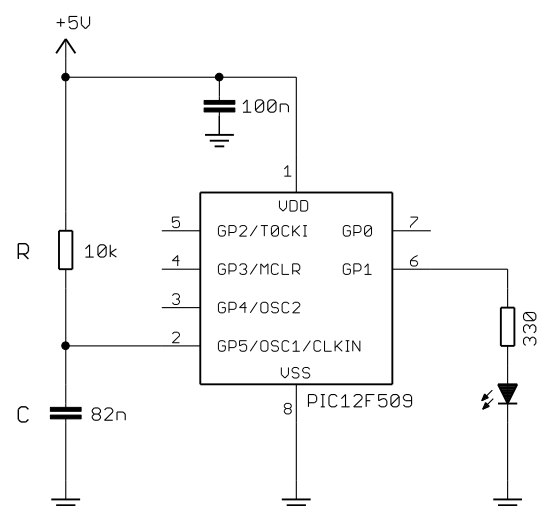
```

External RC oscillator

Finally, a low-cost, low-power option: the baseline PICs can be made to oscillate with timing derived from an external resistor and capacitor, as shown on the right.

To implement this circuit using the [Gooligum baseline training board](#), move the shunt to position 1 (“RC”) of JP20, connecting the 10 kΩ resistor and 82 nF capacitor to OSC1. Remove the shunt from JP21 and leave JP12 closed, enabling the LED on GP1.

External RC oscillators can be appropriate when a very low clock rate can be tolerated – drawing significantly less power than when the internal 4 MHz RC oscillator is used.



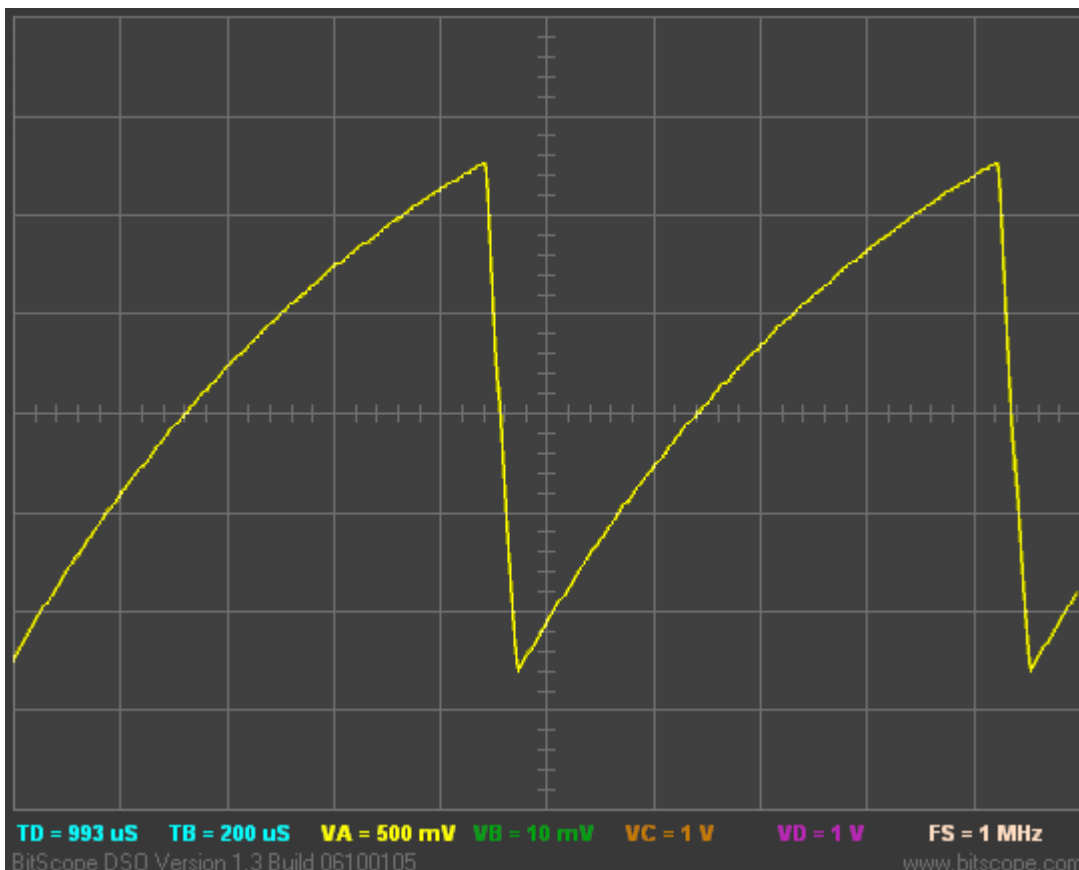
It can also simplify some programming tasks when the PIC is run slowly, needing fewer, shorter delay loops.

The external RC oscillator is a *relaxation* type. The capacitor is charged through the resistor, the voltage v at the OSC1 pin rising with time t according to the formula:

$$v = V_{DD} \left(1 - e^{-t/RC} \right)$$

The voltage increases until it reaches a threshold, typically $0.75 \times V_{DD}$. A transistor is then turned on, which quickly discharges the capacitor until the voltage falls to approx. $0.25 \times V_{DD}$. The capacitor then begins charging through the resistor again, and the cycle repeats.

This is illustrated by the following oscilloscope trace, recorded at the OSC1 pin in the circuit above, with the component values shown:



In theory, assuming upper and lower thresholds of $0.75 \times V_{DD}$ and $0.25 \times V_{DD}$, the period of oscillation is equal to $1.1 \times RC$ (in seconds, with R in Ohms and C in Farads).

In practice, the capacitor discharge is not instantaneous (and of course it can never be), so the period is a little longer than this. Microchip does not commit to a specific formula for the frequency (or period) of the external RC oscillator, only stating that it is a function of V_{DD} , R, C and temperature, and in some documents providing some reference charts. But for rough design guidance, you can assume the period of oscillation is approximately $1.2 \times RC$.

Microchip recommends keeping R between 5 k Ω and 100 k Ω , and C above 20 pF.

In the circuit above, R = 10 k Ω and C = 82 nF. Those values will give a period of approximately:

$$1.2 \times 10 \times 10^3 \times 82 \times 10^{-9} \text{ s} = 984 \mu\text{s}$$

Inverting that gives 1016 Hz.

In practice, the measured frequency was 1052 Hz; reasonably close, but the lesson should be clear: don't use an external RC oscillator if you want high accuracy or good stability.

Only use an external RC oscillator if the exact clock rate is unimportant.

So, given a roughly 1 kHz clock, what can we do with it? Flash an LED, of course!

Using a similar approach to before, we can use the instruction clock (approx. 256 Hz) to increment Timer0. In fact, with a prescale ratio of 1:256, TMR0 will increment at approx. 1 Hz.

TMR0<0> would then cycle at 0.5 Hz, TMR0<1> at 0.25 Hz, etc.

Now consider what happens when the prescale ratio is set to 1:64. TMR0 will increment at 4 Hz, TMR0<0> will cycle at 2 Hz, and TMR0<1> will cycle at 1 Hz, etc.

And that suggests a very simple way to make the LED on GP1 flash at 1Hz. If we continually copy TMR0 to GPIO, each bit of GPIO will continually reflect each corresponding bit of TMR0. In particular, GPIO<1> will always be set to the same value as TMR0<1>. Since TMR0<1> is cycling at 1 Hz, GPIO<1> (and hence GP1) will also cycle at 1 Hz.

Complete program

The following program implements the approach described above. Note that the external RC oscillator is selected by using the option `_ExtRC_OSC` in the configuration statement.

The “main loop” is only three instructions long – by far the shortest “flash an LED” program we have done so far, illustrating how a slow clock rate can sometimes simplify some programming problems.

On the other hand, it is also the least accurate of the “flash an LED” programs, being only approximately 1 Hz. But for many applications, the exact speed doesn't matter; it only matters that the LED visibly flashes.

```

;*****
;
; Description:      Lesson 7, example 4d
;
; Demonstrates use of PIC oscillator in external RC mode (~1 kHz)
;
; LED on GP1 flashes at approx 1 Hz (50% duty cycle),
; with timing derived from instruction clock
;
;*****
;
; Pin assignments:
;   GP1 = flashing LED
;   OSC1 = R (10k) / C (82n)
;
;*****

list      p=12F509
#include  <p12F509.inc>

radix    dec

;***** CONFIGURATION
;           ; ext reset, no code protect, no watchdog, external RC osc
__CONFIG  _MCLRE_ON & _CP_OFF & _WDT_OFF & _ExtRC_OSC

;***** RC CALIBRATION

```

```

RCCAL    CODE    0x3FF                ; processor reset vector
         res 1                ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET    CODE    0x000                ; effective reset vector
         movwf   OSCCAL            ; apply internal RC factory calibration

;***** MAIN PROGRAM *****

;***** Initialisation
start
        ; configure port
        movlw   b'111101'            ; configure GP1 (only) as output
        tris   GPIO
        ; configure timer
        movlw   b'11010101'         ; configure Timer0:
        ; --0-----                timer mode (T0CS = 0)
        ; ----0----                prescaler assigned to Timer0 (PSA = 0)
        ; -----101              prescale = 64 (PS = 101)
        option                ; -> increment at 4 Hz with 1 kHz clock

;***** Main loop
main_loop
        ; TMR0<1> cycles at 1 Hz, so continually copy to LED (GP1)
        movf    TMR0,w              ; copy TMR0 to GPIO
        movwf   GPIO

        ; repeat forever
        goto   main_loop

        END

```

Conclusion

That completes our coverage of the PIC12F509.

We've seen that baseline PICs can be put into a low-power sleep mode, and that they can be configured to be woken by an external event (a pin change), or on a regular basis by the watchdog timer, which is also (in fact, primarily) useful for restarting the device if it gets “stuck”, following some type of error condition.

We also saw that PICs can be clocked in a number of ways, that there are alternatives to the internal RC oscillator, such as an external clock, and the PIC's own oscillator circuitry driving a crystal, resonator, or a simple RC timing circuit – and we discussed some of the reasons for using those alternatives.

To introduce further topics, we need a larger device. In the [next lesson](#) we'll move onto the 14-pin 16F506, and see how to use lookup tables (and those extra pins) to drive 7-segment displays, and how to use multiplexing to drive multiple displays.

And it will be nice to finally get away from flashing LEDs for a while!

Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 8: Driving 7-Segment Displays

The applications we've looked at so far have used only one or two LEDs as outputs. That's enough for simple indicators, but many applications need to be able to display information in numeric, alphanumeric or graphical form. Although LCD and OLED displays are becoming more common, there is still a place, when displaying numeric (or sometimes hexadecimal) information, for 7-segment LED displays.

To drive a single 7-segment display, in a straightforward manner, we need seven outputs. That rules out the PIC12F509 we've been using so far. Its bigger brother, the 14-pin 16F505, is quite suitable, but to avoid using too many different devices, we'll jump to the more capable 16F506. In fact, the 16F506 can be made to drive up to four 7-segment displays, using a technique known as *multiplexing*. But to display even a single digit, that digit has to be translated into a specific pattern of segments in the display. That translation is normally done through *lookup tables*.

In summary, this lesson covers:

- Introductory overview of the PIC16F506 MCU
- Driving a single 7-segment display
- Using lookup tables
- Using multiplexing to drive multiple displays
- Binary-coded decimal (BCD)

Introducing the PIC16F506

The previous lessons have focussed on the 10F200 (or 12F508) and 12F509.

We saw in [lesson 1](#) that the 12F508 and 12F509 are part of a family which includes the 14-pin 16F505. That lesson included the following table, summarising the differences within the 12F508/12F509/16F505 family:

Device	Program Memory (words)	Data Memory (bytes)	Package	I/O pins	Clock rate (maximum)
12F508	512	25	8-pin	6	4 MHz
12F509	1024	41	8-pin	6	4 MHz
16F505	1024	72	14-pin	12	20 MHz

Although the 16F505 is architecturally very similar to the 12F508/509, it has more data memory, more I/O pins (11 I/O and 1 input-only), a higher maximum clock speed and wider range of oscillator options.

The 12F510 and 16F506 form a very similar family, adding peripherals with *analog* (continuously variable) inputs: analog comparators and an analog-to-digital converter (ADC). We'll explore those capabilities in

lessons [9](#) and [10](#), but briefly – a comparator allows us to compare two analog signals (one of which is often a fixed reference voltage), while the ADC allows us to measure analog signals.

The following table compares the features of the devices in both families:

Device	Program Memory (words)	Data Memory (bytes)	Package	I/O pins	Comparators	Analog Inputs	Clock rate (maximum)
12F508	512	25	8-pin	6	-	-	4 MHz
12F509	1024	41	8-pin	6	-	-	4 MHz
12F510	1024	38	8-pin	6	1	3	8 MHz
16F505	1024	72	14-pin	12	-	-	20 MHz
16F506	1024	67	14-pin	12	2	3	20 MHz

Although the 16F505 would be adequate for this lesson, we may as well jump directly to the 16F506, which does everything the 16F505 does (although it does have 5 bytes less data memory...) and continue to use it when we look at analog inputs in the upcoming lessons. We'll just ignore the analog side for now.

The expanded capabilities of the 16F506 (other than analog) are detailed in the following sections.

Additional oscillator options

The 16F506 supports an expanded range of oscillator options, selected by bits in the configuration word:

Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	-	-	IOSCFS	MCLRE	$\overline{\text{CP}}$	WDTE	FOSC2	FOSC1	FOSC0

In the 12F510/16F506 devices, the internal RC oscillator can optionally run at a nominal 8 MHz instead of 4 MHz. *Be careful, if you select 8 MHz, that any code (such as delays) written for a 4 MHz clock is correct.*

The speed of the internal RC oscillator is selected by the IOSCFS bit.

Setting IOSCFS to '1' (by ANDing the symbol '`_IOSCFS_ON`' into the configuration word expression) selects 8 MHz operation; clearing it to '0' (with '`_IOSCFS_OFF`') selects 4 MHz.

The three FOSC bits allow the selection of eight clock options (twice the number available in the 12F509), as in the table below.

FOSC<2:0>	Standard symbol	Oscillator configuration
000	<code>_LP_OSC</code>	LP oscillator
001	<code>_XT_OSC</code>	XT oscillator
010	<code>_HS_OSC</code>	HS oscillator
011	<code>_EC_RB4EN</code>	EC oscillator + RB4
100	<code>_IntRC_OSC_RB4EN</code>	Internal RC oscillator + RB4
101	<code>_IntRC_OSC_CLKOUTEN</code>	Internal RC oscillator + CLKOUT
110	<code>_ExtRC_OSC_RB4EN</code>	External RC oscillator + RB4
111	<code>_ExtRC_OSC_CLKOUTEN</code>	External RC oscillator + CLKOUT

The 'LP' and 'XT' oscillator options are exactly the same as described in [lesson 7](#): 'LP' mode being typically used to drive crystals with a frequency less than 200 kHz, and 'XT' mode being intended for crystals or resonators with a frequency between 200 kHz and 4 MHz.

The ‘HS’ (“high speed”) mode extends this to 20 MHz. The crystal or resonator, with appropriate loading capacitors, is connected between the OSC1 and OSC2 pins in exactly the same way as for the ‘LP’ or ‘XT’ modes.

As explained in [lesson 7](#), the ‘LP’ and ‘XT’ (and indeed ‘HS’) modes can be used with an external clock signal, driving the OSC1, or CLKIN, pin. The downside to using the “crystal” modes with an external clock is that the OSC2 pin remains unused, wasting a potentially valuable I/O pin.

The ‘EC’ oscillator mode addresses this problem. It is designed for use with an external clock signal driving the CLKIN pin, the same as is possible in the crystal modes, but with the significant advantage that the “OSC2 pin”, pin 3 on the 16F506, is available for digital I/O as pin ‘RB4’.

There are now two internal RC oscillator modes. ‘_IntRC_OSC_RB4EN’ is just like the 12F509’s ‘_IntRC_OSC’ mode, where the internal RC oscillator runs (at either 4 MHz or 8 MHz on the 16F506) leaving all pins available for digital I/O – including RB4 (pin 3).

The second internal RC option, ‘_IntRC_OSC_CLKOUTEN’, assigns pin 3 as ‘CLKOUT’ instead of RB4. In this mode, the instruction clock, which runs at one quarter the speed of the processor clock, i.e. a nominal 1 MHz (or 2 MHz if IOSCFS is set), is output on the CLKOUT pin. This output clock signal can be used to provide a clock signal to external devices, or for synchronising other devices with the PIC.

[Lesson 7](#) showed how an external RC oscillator can be used with the 12F509. Although this mode usefully allows for low cost, low power operation, it has the same drawback as the externally-clocked “crystal” modes: pin 3 (OSC2) cannot be used for anything.

The external RC oscillator modes on the 16F506 overcome this drawback. In the first option, ‘_ExtRC_OSC_RB4EN’, pin 3 is available for digital I/O as RB4.

The other external RC option, ‘_ExtRC_OSC_CLKOUTEN’, assigns pin 3 to CLKOUT, with the instruction clock appearing as an output signal, running at one quarter the rate of the external RC oscillator (FOSC/4).

In summary, the expanded range of clock options provides for higher speed operation, more usable I/O pins, or a clock output to allow for external device synchronisation.

Additional I/O pins

The 16F506 provides twelve I/O pins (one being input-only), compared with the six (with one being input-only) available on the 12F508/509/510.

Twelve is too many pins to represent in a single 8-bit register, so instead of a single port named GPIO, the 16F506 has two ports, named PORTB and PORTC.

Six I/O pins are allocated to each port:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTB			RB5	RB4	RB3	RB2	RB1	RB0
PORTC			RC5	RC4	RC3	RC2	RC1	RC0

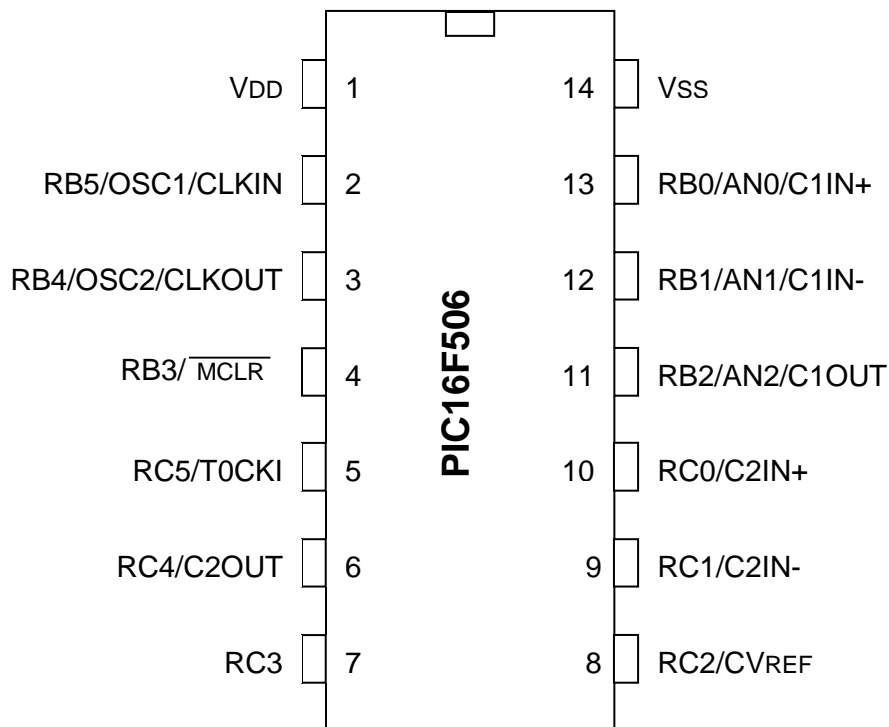
The direction of each I/O pin is controlled by corresponding TRIS registers:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TRISB			RB5	RB4		RB2	RB1	RB0
TRISC			RC5	RC4	RC3	RC2	RC1	RC0

As in the 12F509, the TRIS registers are not mapped into data memory and can only be accessed through the 'tris' instruction, with an operand of 6 (or 'PORTB') to load TRISB, or an operand of 7 (or 'PORTC') to load TRISC.

RB3 is input only and, like GP3 on the 12F509, it shares a pin with $\overline{\text{MCLR}}$; the pin assignment being controlled by the MCLRE bit in the configuration word.

The 16F506 comes in a 14-pin package; the pin diagram is shown below.



Note that RC5 and T0CKI (the Timer0 external clock input) share the same pin.

We have seen that on the 12F509, T0CKI shares a pin with GP2, and to use GP2 as an output you must first disable T0CKI by clearing the T0CS bit in the OPTION register.

In the same way, to use RC5 as an output on the 16F506, you must first disable T0CKI by clearing T0CS.

The RB0, RB1 and RB2 pins are configured as analog inputs by default. To use any of these pins for digital I/O, they must be deselected as analog inputs. This can be done by clearing the ADCON0 register, as we'll see in [lesson 10](#) on analog-to-digital conversion.

The RB0, RB1, RC0 and RC1 pins are configured as comparator inputs by default. To use any of these pins for digital I/O, the appropriate comparator must be disabled (by clearing the C1ON bit in the CM1CON0 register, and/or the C2ON bit in the CM2CON0 register), or its inputs reassigned, as explained in [lesson 9](#).

Note: On PICs with comparators and/or analog (ADC) inputs, the comparator and analog inputs are enabled on start-up. To use a pin for digital I/O, any comparator or analog input assigned to that pin must first be disabled.

This is a common trap for beginners, who wonder why their LED won't light, when they haven't deselected analog input on the pin they are using. That is why this tutorial series began with digital-only PICs.

For now, we'll just include the instructions to disable these analog inputs in the examples in this lesson, and leave the full explanations for lessons [9](#) and [10](#).

Additional data memory

The data memory, or register file, of the 16F506 is arranged in four banks, as follows:

PIC16F506 Registers

Bank 0		Bank 1		Bank 2		Bank 3	
00h	INDF	20h	INDF	40h	INDF	60h	INDF
01h	TMR0	21h	TMR0	41h	TMR0	61h	TMR0
02h	PCL	22h	PCL	42h	PCL	62h	PCL
03h	STATUS	23h	STATUS	43h	STATUS	63h	STATUS
04h	FSR	24h	FSR	44h	FSR	64h	FSR
05h	OSCCAL	25h	OSCCAL	45h	OSCCAL	65h	OSCCAL
06h	PORTB	26h	PORTB	46h	PORTB	66h	PORTB
07h	PORTC	27h	PORTC	47h	PORTC	67h	PORTC
08h	CM1CON0	28h	CM1CON0	48h	CM1CON0	68h	CM1CON0
09h	ADCON0	29h	ADCON0	49h	ADCON0	69h	ADCON0
0Ah	ADRES	2Ah	ADRES	4Ah	ADRES	6Ah	ADRES
0Bh	CM2CON0	2Bh	CM2CON0	4Bh	CM2CON0	6Bh	CM2CON0
0Ch	VRCON	2Ch	VRCON	4Ch	VRCON	6Ch	VRCON
0Dh	Shared GP Registers	2Dh	Map to Bank 0 0Dh – 0Fh	4Dh	Map to Bank 0 0Dh – 0Fh	6Dh	Map to Bank 0 0Dh – 0Fh
0Fh		2Fh		4Fh		6Fh	
10h	General Purpose Registers	30h	General Purpose Registers	50h	General Purpose Registers	70h	General Purpose Registers
1Fh		3Fh		5Fh		7Fh	

There are only 3 shared data registers (0Dh – 0Fh), which are mapped into all four banks.

In addition, there are $4 \times 16 = 64$ non-shared (*banked*) data registers, filling the top half of each bank.

Thus, the 16F506 has a total of $3 + 64 = 67$ general purpose data registers.

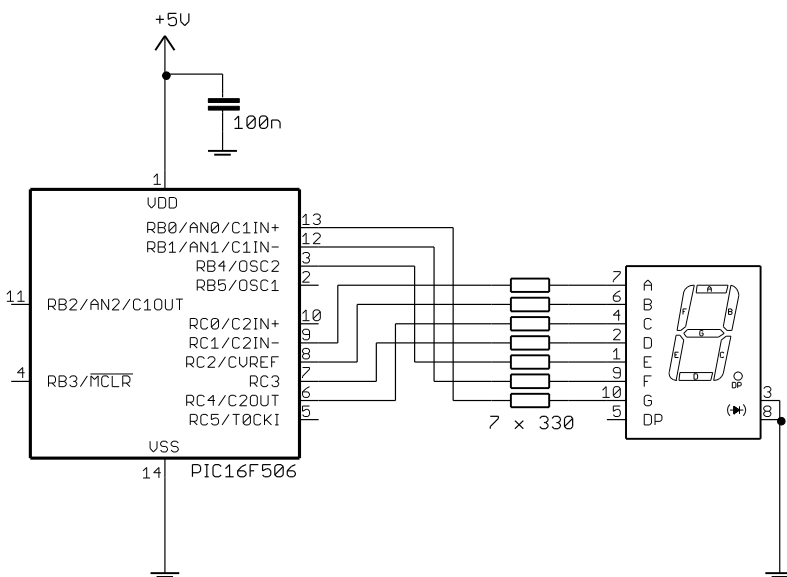
The bank is selected by the FSR<6:5> bits, as was explained (for the 16F505) in [lesson 3](#). Although an additional bank selection bit is used, compared with the single bit in the 12F509, you don't need to be aware of that; simply use the `banksel` directive in the usual way.

Driving a 7-segment LED Display

A 7-segment LED display is simply a collection of LEDs, typically one per segment (but often having two or more LEDs per segment for large displays), arranged in the “figure 8” pattern we are familiar with from numeric digital displays. 7-segment display modules also commonly include one or two LEDs for decimal points.

7-segment LED display modules come in one of two varieties: common-anode or common-cathode.

In a common-cathode module, the cathodes belonging to each segment are wired together within the module, and brought out through one or two (or sometimes more) pins. The anodes for each segment are brought out separately, each to its own pin. Typically, each segment (anode) would be connected to a separate output pin on the PIC, as shown in the following circuit diagram¹:



The common cathode pins are connected together and grounded.

To light a given segment in a common-cathode display, the corresponding PIC output is set high. Current flows from the output and through the given segment (limited by a series resistor) to ground.

In a common-anode module, this is reversed; the anodes for each segment are wired together and the cathodes are accessible separately. In that case, the common anode pins are connected to the positive supply and each cathode is connected to a separate PIC output.

To light a segment in a common-anode display, the corresponding PIC output is set low; current flows from the positive supply, through the segment and into the PIC's output.

Although, on the PIC16F506, a single pin can source or sink up to 25 mA, the maximum per port is 100 mA and the maximum current into VDD (the device's supply current) is 150 mA. Given that the PIC itself consumes some current (up to around 2 mA) and that we'd potentially like to be able to draw current from the unused output pins, we should limit the total current drawn by the 7-segment display to no more than 100 mA or so. Since all the segments may be lit at once (when displaying '8'), we should limit the current per pin to $100\text{ mA} \div 7 = 14.3\text{ mA}$. The $330\ \Omega$ resistors limit the current to 10 mA, well within spec while giving a bright display.

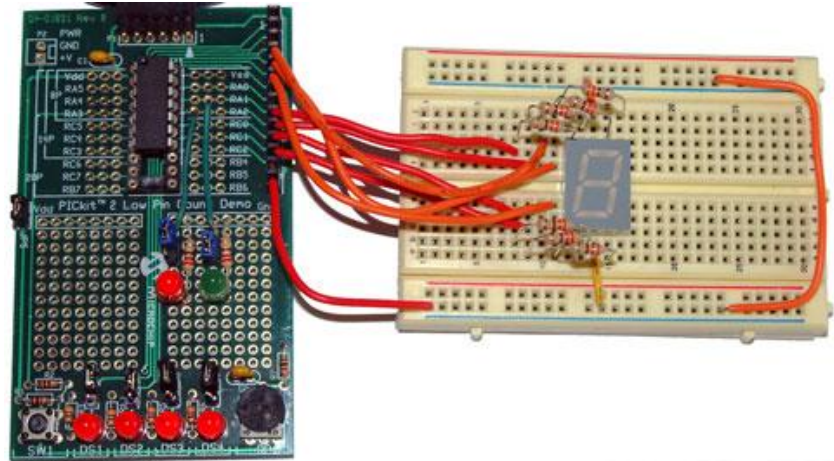
If you are using the [Gooligum baseline training board](#), you can implement this circuit by:

- placing shunts (six of them) across every position in jumper block JP4, connecting segments A-D, F and G to pins RB0-1 and RC1-4
- placing a single shunt in position 1 (“RA/RB4”) of JP5, connecting segment E to pin RB4
- placing a shunt across pins 1 and 2 (“GND”) of JP6, connecting digit 1 to ground.

All other shunts should be removed.

¹ The segment anodes are connected to PIC pins in the (apparently) haphazard way shown, because this reflects the connections on the [Gooligum baseline training board](#). You'll often find that, by rearranging your PIC pin assignments, you can simplify your PCB layout and routing – even if it makes your schematic messier!

If you are using Microchip's Low Pin Count Demo Board, you will have to supply your own 7-segment display module, and connect it (and the current-limiting resistors) to the board. This can be done via the 14-pin header on the Low Pin Count Demo Board, as illustrated on the right. Note that the header pins corresponding to the "RB" pins on the 16F506 are labelled "RA" on the demo board, reflecting the PIC16F690 it is supplied with, not the 16F506 used here.



Be careful, because your 7-segment display module may have a different pin-out to that shown above. If you have a common-anode display, you will need to wire it correctly and make appropriate changes to the code presented here, but the techniques for driving the display are essentially the same.

Lookup tables

To display each digit, a corresponding pattern of segments must be lit, as follows:

Segment:	A	B	C	D	E	F	G
Pin:	RC1	RC2	RC4	RC3	RB4	RB1	RB0
0	on	on	on	on	on	on	off
1	off	on	on	off	off	off	off
2	on	on	off	on	on	off	on
3	on	on	on	on	off	off	on
4	off	on	on	off	off	on	on
5	on	off	on	on	off	on	on
6	on	off	on	on	on	on	on
7	on	on	on	off	off	off	off
8	on	on	on	on	on	on	on
9	on	on	on	on	off	on	on

We need a way to determine, or look up, the pattern corresponding to the digit to be displayed, and that is most effectively done with a *lookup table*.

The most common method of implementing lookup tables in the baseline PIC architecture is to use a *computed jump* into a table of 'retlw' instructions.

For example, to look up the binary pattern to be applied to PORTC, corresponding to the digit in W, we could use the following subroutine:

```
; pattern table for 7 segment display on port C
; RC4:1 = CDBA
get7sC  addwf  PCL,f
        retlw b'011110'    ; 0
        retlw b'010100'    ; 1
        retlw b'001110'    ; 2
        retlw b'011110'    ; 3
        retlw b'010100'    ; 4
        retlw b'011010'    ; 5
        retlw b'011010'    ; 6
        retlw b'010110'    ; 7
        retlw b'011110'    ; 8
        retlw b'011110'    ; 9
```

Baseline PICs have a single addition instruction: ‘addwf f,d’ – “add W to file register”, placing the result in the register if the destination is ‘,f’, or in W if the destination is ‘,w’.

As mentioned in [lesson 3](#), the program counter (PC) is a 12-bit register holding the full address of the next instruction to be executed. The lower eight bits of the program counter (PC<7:0>) are mapped into the PCL register. If you change the contents of PCL, you change the program counter – affecting which instruction will be executed next. For example, if you add 2 to PCL, the program counter will be advanced by 2, skipping the next two instructions.

In the code above, the first instruction adds the table index, or offset (corresponding to the digit being looked up), in W to PCL, writing the result back to PCL.

If W contains ‘0’, 0 is added to PCL, leaving the program counter unchanged, and the next instruction is executed as normal: the first ‘retlw’, returning the pattern for digit ‘0’ in W.

But consider what happens if the subroutine is called with W containing ‘4’. PCL is incremented by 4, advancing the program counter by 4, so the next four instructions will be skipped. The fifth ‘retlw’ instruction will be executed, returning the pattern for digit ‘4’ in W.

This lookup table could then be used (‘called’, since it is actually a subroutine) as follows:

```
movf    digit,w           ; get digit to display
call    get7sC           ; lookup pattern for port C
movwf   PORTC            ; then output it
```

(assuming that the digit to be displayed is stored in a variable called ‘digit’)

A second lookup table, called the same way, would be used to lookup the pattern to be output on PORTB.

The define table directive

Since lookup tables are very useful, and commonly used, the MPASM assembler provides a shorthand way to define them: the ‘dt’ (short for “define table”) directive. Its syntax is:

```
[label] dt      expr1[,expr2,...,exprN]
```

where each expression is an 8-bit value. This generates a series of retlw instructions, one for each expression. The directive is equivalent to:

```
[label] retlw  expr1
        retlw  expr2
        ...
        retlw  exprN
```

Thus, we could write the code above as:

```
get7sC  addwf   PCL, f
        dt     b'011110',b'010100',b'001110',b'011110',b'010100' ; 0,1,2,3,4
        dt     b'011010',b'011010',b'010110',b'011110',b'011110' ; 5,6,7,8,9
```

or it could even be written as:

```
get7sC  addwf   PCL, f
        dt     0x1E,0x14,0x0E,0x1E,0x14,0x1A,0x1A,0x16,0x1E,0x1E ; 0-9
```

Of course, the `dt` directive is more appropriate in some circumstances than others. Your table may be easier to understand if you use only one expression per line, in which case it is clearer to simply use `retlw`.

A special case where ‘`dt`’ makes your code much more readable is with text strings. For example:

```
dt      "Hello world",0
```

is equivalent to:

```
retlw   'H'
retlw   'e'
retlw   'l'
retlw   'l'
retlw   'o'
retlw   ' '
retlw   'w'
retlw   'o'
retlw   'r'
retlw   'l'
retlw   'd'
retlw   0
```

The ‘`dt`’ form is clearly preferable in this case.

Lookup table address limitation

A significant limitation of the baseline PIC architecture is that, when any instruction modifies `PCL`, bit 8 of the program counter (`PC<8>`) is cleared. That means that, whatever the result of the table offset addition, when `PCL` is updated, the program counter will be left pointing at an address in the first 256 words of the current program memory page (`PC<9>` is updated from the `PA0` bit, in the same way as for a `goto` or `call` instruction; see [lesson 3](#).)

This is very similar to the address limitation, discussed in [lesson 3](#), which applies to subroutines on baseline PICs. But the constraint on lookup tables is even more limiting – because the result of the offset addition must be within the first 256 words of a page, not just the start of the table, the whole table has to fit within the first 256 words of a page.

In the baseline PIC architecture, lookup tables must be wholly contained within the first 256 locations of a program memory page.

We have seen that a workaround for the limitation on subroutine addressing is to use a vector table, but no such workaround is possible for lookup tables.

Therefore you must take care to ensure that any lookup tables are located toward the beginning of a program memory page. A simple way to do that is to place the lookup tables in a separate code section, located explicitly at the start of a page, by specifying its address with the `CODE` directive.

For example:

```
TABLES CODE 0x200 ; locate at beginning of a page

; pattern table for 7 segment display on port B
; RB4 = E, RB1:0 = FG
get7sB addwf PCL,f
      retlw b'010010' ; 0
      retlw b'000000' ; 1
      retlw b'010001' ; 2
      retlw b'000001' ; 3
      retlw b'000011' ; 4
      retlw b'000011' ; 5
      retlw b'010011' ; 6
      retlw b'000000' ; 7
      retlw b'010011' ; 8
      retlw b'000011' ; 9
```

This places the tables explicitly at the beginning of page 1 (the 16F506 has two program memory pages), out of the way of the start-up code located at the beginning of page 0 (0x000).

This means of course that you need to use the `pagesel` directive if calling these lookup tables from a different code section.

To display a digit, we need to look up and then write the correct patterns for ports B and C, meaning two table lookups for each digit displayed.

Ideally we'd have a single routine which, given the digit to be displayed, performs the table lookups and writes the patterns to the I/O ports. To avoid the need for multiple `pagesel` directives, this "display digit" subroutine can be located on the same page as the lookup tables.

For example:

```
TABLES CODE 0x200 ; locate at beginning of a page

; pattern table for 7 segment display on port B
; RB4 = E, RB1:0 = FG
get7sB addwf PCL,f
      retlw b'010010' ; 0
      retlw b'000000' ; 1
      ... (etc.)

; pattern table for 7 segment display on port C
; RC4:1 = CDBA
get7sC addwf PCL,f
      retlw b'011110' ; 0
      retlw b'010100' ; 1
      ... (etc.)

; Display digit passed in 'digit' variable on 7-segment display
set7seg_R
      movf digit,w ; get digit to display
      call get7sB ; lookup pattern for port B
      movwf PORTB ; then output it
      movf digit,w ; repeat for port C
      call get7sC
      movwf PORTC
      retlw 0
```

Then to display a digit, it is simply a matter of writing the value into the 'digit' variable (assumed to be in a shared data segment to avoid the need for banking), and calling the 'set7seg_R' routine.

Note that it's assumed that the 'set7seg_R' routine is called through a vector in page 0 labelled 'set7seg', so that the subroutine doesn't have to be in the first 256 words of page 1; it can be anywhere on page 1 and we still avoid the need for a 'pagesel' when calling the lookup tables from it.

So, given these lookup tables and a subroutine that will display a selected digit, what to do with them? We've been blinking LEDs at 1 Hz, so counting seconds seems appropriate.

Complete program

The following program incorporates the code fragments presented above, and code (e.g. macros) and techniques from previous lessons, to count repeatedly from 0 to 9, with 1 s between each count.

```

;*****
; Description: Lesson 8, example 1a *
; *
; Demonstrates use of lookup tables to drive 7-segment display *
; *
; Single digit 7-segment LED display counts repeating 0 -> 9 *
; 1 second per count, with timing derived from int 4 MHz oscillator *
; *
;*****
; Pin assignments: *
; RB0-1, RB4, RC1-4 = 7-segment display bus (common cathode) *
; *
;*****

list      p=16F506
#include   <p16F506.inc>

#include   <stdmacros-base.inc>      ; DelayMS - delay in milliseconds
                                           ; (calls delay10)
EXTERN    delay10_R                 ; W x 10 ms delay

radix     dec

;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IOSCF5_OFF & _IntRC_OSC_RB4EN

;***** VARIABLE DEFINITIONS
                UDATA_SHR
digit       res 1                    ; digit to be displayed

;***** RC CALIBRATION
RCCAL       CODE    0x3FF             ; processor reset vector
                res 1                 ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET       CODE    0x000             ; effective reset vector
                movwf  OSCCAL          ; apply internal RC factory calibration
                pagesel start
                goto   start          ; jump to main code

```

```

;***** Subroutine vectors
delay10                                ; delay W x 10 ms
    pagesel delay10_R
    goto    delay10_R
set7seg                                  ; display digit on 7-segment display
    pagesel set7seg_R
    goto    set7seg_R

;***** MAIN PROGRAM *****
MAIN    CODE

;***** Initialisation
start
    ; configure ports
    clr    PORTB                        ; configure PORTB and PORTC as all outputs
    tris   PORTB
    tris   PORTC
    clrf   ADCON0                       ; disable AN0, AN1, AN2 inputs
    bcf    CM1CON0,C1ON                  ; and comparator 1 -> RB0,RB1 digital
    bcf    CM2CON0,C2ON                  ; disable comparator 2 -> RC1 digital

    ; initialise variables
    clrf   digit                        ; start with digit = 0

;***** Main loop
main_loop
    ; display digit
    pagesel set7seg
    call   set7seg

    ; delay 1 sec
    DelayMS 1000

    ; increment digit
    incf   digit,f
    movlw  .10
    xorwf  digit,w                       ; if digit = 10
    btfsc  STATUS,Z
    clrf   digit                        ; reset it to 0

    ; repeat forever
    pagesel main_loop
    goto   main_loop

;***** LOOKUP TABLES *****
TABLES CODE    0x200                    ; locate at beginning of a page

; pattern table for 7 segment display on port B
; RB4 = E, RB1:0 = FG
get7sB  addwf  PCL,f
        retlw  b'010010'                ; 0
        retlw  b'000000'                ; 1
        retlw  b'010001'                ; 2
        retlw  b'000001'                ; 3
        retlw  b'000011'                ; 4
        retlw  b'000011'                ; 5
        retlw  b'010011'                ; 6
        retlw  b'000000'                ; 7
        retlw  b'010011'                ; 8
        retlw  b'000011'                ; 9

```

```

; pattern table for 7 segment display on port C
; RC4:1 = CDBA
get7sC  addwf   PCL,f
        retlw  b'011110'      ; 0
        retlw  b'010100'      ; 1
        retlw  b'001110'      ; 2
        retlw  b'011110'      ; 3
        retlw  b'010100'      ; 4
        retlw  b'011010'      ; 5
        retlw  b'011010'      ; 6
        retlw  b'010110'      ; 7
        retlw  b'011110'      ; 8
        retlw  b'011110'      ; 9

; Display digit passed in 'digit' variable on 7-segment display
set7seg_R
        movf   digit,w        ; get digit to display
        call  get7sB          ; lookup pattern for port B
        movwf PORTB           ; then output it
        movf   digit,w        ; repeat for port C
        call  get7sC
        movwf PORTC
        retlw  0

        END

```

Multiplexing

To display multiple digits, as in (say) a digital clock, the obvious approach is to extend the method we've just used for a single digit. That is, where one digit requires 7 outputs, two digits would apparently need 14 outputs; four digits would need 28 outputs, etc.

At that rate, you would very quickly run out of output pins, even on the bigger PICs!

A technique commonly used to conserve pins is to *multiplex* a number of displays (and/or inputs – a topic we'll look at another time).

When displays are multiplexed, only one (or a subset) of them is on at any time.

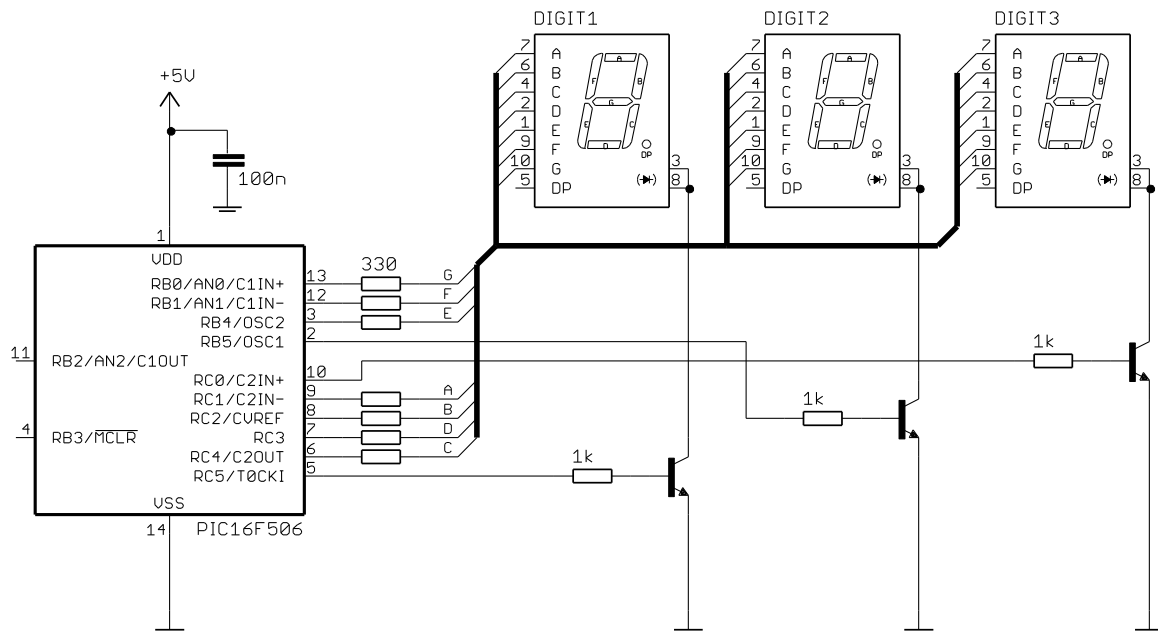
Display multiplexing relies on speed, and human persistence of vision, to create an illusion that a number of displays are on at once, whereas in fact they are being lit rapidly in sequence, so quickly that it appears that they are on continuously.

To multiplex 7-segment displays, it is usual to connect each display in parallel, so that one set of output pins on the PIC is connected to every display, the connections between the modules and to the PIC forming a *bus*.

If the common cathodes were all grounded, every module would display the same digit (feebly, since the output current would be shared between them).

Instead, to allow a different digit to be displayed on each module, the individual displays must be capable of being switched on or off under software control.

For that, transistors are usually used as switches, as illustrated below²:



Note that it is not possible to connect the common cathodes directly to the PIC's pins; the combined current from all the segments in a module will be up to 70 mA – too high for a single pin to sink. Instead, the pin is used to switch a transistor on or off.

Almost any low-cost NPN transistor³, such as a BC547, could be used for this, as is it not a demanding application. It's also possible to use FETs; for example, MOSFETs are usually used to switch high-power devices.

When the output pin is set 'high', the transistor's base is pulled high, turning it 'on'. The 1 kΩ resistors are used to limit the base current to around 4 mA – enough to saturate the transistor, effectively grounding the module's common cathode connection, allowing the display connected to that transistor to light.

These transistors are then used to switch each module on, in sequence, for a short time, while the pattern for that digit is output on the display bus. This is repeated for each digit in the display, quickly enough to avoid visible flicker (preferably at least 70 times per second).

To implement this circuit using the [Gooligum baseline training board](#):

- keep the six shunts in every position of jumper block JP4, connecting segments A-D, F and G to pins RB0-1 and RC1-4
- keep the shunt in position 1 ("RA/RB4") of JP5, connecting segment E to pin RB4
- move the shunt in JP6 to across pins 2 and 3 ("RC5"), connecting digit 1 to the transistor controlled by RC5
- place shunts in jumpers JP8, JP9 and JP10, connecting pins RC5, RB5 and RC0 to their respective transistors

All other shunts should be removed.

² Again, the diagram reflects the connections on the [Gooligum baseline training board](#), not what looks neatest.

³ If you had common-anode displays, you would normally use PNP transistors as high-side switches (between VDD and each common anode), instead of the NPN low-side switches shown here.

Example application

To demonstrate display multiplexing, we'll implement a simple timer: the first digit will count minutes and the next two digits will count seconds (00 to 59).

The approach taken in the single-digit example above – set the outputs and then delay for 1 s – won't work, because the display multiplexing has to continue throughout the delay.

Ideally the display multiplexing would be a “background task”; one that continues steadily while the main program is free to perform tasks such as responding to changing inputs. That's an ideal application for timer-based interrupts – a feature available on more advanced PICs (as we will see in [midrange lesson 12](#)), but not baseline devices like the 16F506.

But a timer can still be used to good advantage when implementing multiplexing on a baseline PIC. It would be impractical to try to use programmed delays while multiplexing; there's too much going on. But Timer0 can provide a steady *tick* that we can base our timing on – displaying each digit for a single tick, and then counting ticks to decide when a certain time (e.g. 1 sec) has elapsed and we need to perform an action (such as incrementing counters).

If the tick period is too short, there may not be enough time to complete all the program logic needed between ticks, but if it's too long, the display will flicker.

Many PIC developers use a standard 1 ms tick, but to simplify the task of counting in seconds, an (approximately) 2 ms tick is used in this example. If each of three digits is updated at a rate of 2 ms per digit, the whole 3-digit display is updated every 6 ms, so the display rate is $1 \div 6 \text{ ms} = 167 \text{ Hz}$ – fast enough to avoid perceptible flicker.

To generate an approximately 2 ms tick, we can use Timer0 in timer mode (based on the 1 MHz instruction clock), with a prescale ratio of 1:256. Bit 2 of Timer0 will then be changing with a period of 2048 μs .

In pseudo-code, the multiplexing technique used here is:

```
time = 0:00
repeat
    tick count = 0
    repeat
        display minutes digit for 1 tick (2 ms)
        display tens digit for 1 tick
        display ones digit for 1 tick
    until tick count = #ticks in 1 second

    increment time by 1 second
forever
```

To store the time, the simplest approach is to use three variables, to store the minutes, tens and ones digits separately. Setting the time to zero then means clearing each of these variables.

To display a single digit, such as minutes, the code becomes:

```
        ; display minutes for 2.048 ms
w60_hi  btfss   TMR0,2           ; wait for TMR<2> to go high
        goto   w60_hi
        movf   mins,w           ; output minutes digit
        pagesel set7seg
        call   set7seg
        pagesel $
w60_lo  bsf    MINUTES          ; enable minutes display
        btfsc  TMR0,2           ; wait for TMR<2> to go low
        goto   w60_lo
```

This routine begins by waiting for TMR0<2> to go high, then displays the minutes digit (with the others turned off), and finally waits for TMR0<2> to go low again.

The routine to display the tens digit also begins with a wait for TMR0<2> to go high:

```

                ; display tens for 2.048 ms
w10_hi  btfss   TMR0,2           ; wait for TMR0<2> to go high
        goto    w10_hi
        movf    tens,w           ; output tens digit
        pagesel set7seg
        call    set7seg
        pagesel $
        bsf     TENS             ; enable tens display
w10_lo  btfsc   TMR0,2           ; wait for TMR<2> to go low
        goto    w10_lo

```

There is no need to explicitly turn off the minutes digit, since, whenever a new digit pattern is output by the ‘set7seg’ routine, the “digit enable” pins, RB5, RC0 and RC5 are always cleared (because the digit pattern tables contain ‘0’s for these bits). Thus, all the displays are blanked whenever a new digit is output.

The ones digit is then displayed in the same way:

```

                ; display ones for 2.048 ms
w1_hi   btfss   TMR0,2           ; wait for TMR0<2> to go high
        goto    w1_hi
        movf    ones,w           ; output ones digit
        pagesel set7seg
        call    set7seg
        pagesel $
        bsf     ONES            ; enable ones display
w1_lo   btfsc   TMR0,2           ; wait for TMR<2> to go low
        goto    w1_lo

```

By waiting for TMR0<2> high at the start of each digit display routine, we can be sure that each digit is displayed for exactly 2.048 ms (or, as close as the internal RC oscillator allows, which is only accurate to 1% or so...).

Note that the ‘set7seg’ subroutine has been modified to accept the digit to be displayed as a parameter passed in W, instead of placing it a shared variable; it shortens the code a little to do it this way.

It’s also a good idea to blank the display, by clearing the digit enable lines, before outputting the new digit pattern on the display bus – this avoids “ghosting” (visible in low light) due to PORTB being updated while the pattern for the previous digit is still being output on PORTC.

So the ‘set7seg’ routine becomes:

```

; Display digit passed in W on 7-segment display
set7seg_R
        ; disable displays
        clrf   PORTB           ; clear all digit enable lines on PORTB
        clrf   PORTC           ; and PORTC

        ; output digit pattern
        movwf  temp            ; save digit
        call   get7sB          ; lookup pattern for port B
        movwf  PORTB           ; then output it
        movf   temp,w          ; get digit
        call   get7sC          ; then repeat for port C
        movwf  PORTC
        retlw  0

```

Note also the ‘pagesel \$’ after the subroutine call. It is necessary to ensure that the current page is selected before the ‘goto’ commands in the wait loops are executed.

After TMR0<2> goes low at the end of the ‘ones’ display routine, there is approximately 1 ms before it will go high again, when the ‘minutes’ display will be scheduled to begin again. That means that there is a “spare” 1 ms, after the end of the ‘ones’ routine, in which to perform the program logic of counting ticks and incrementing the time counters; 1 ms is 1000 instruction cycles – plenty of time!

The following code construct continues multiplexing the digit display until 1 second has elapsed:

```
; multiplex display for 1 sec
    movlw    1000000/2048/3    ; display each of 3 digits for 2.048 ms each
    movwf    mpx_cnt          ; repeat multiplex loop for 1 second

mplex_loop
    ; display minutes for 2.048 ms

    ; display tens for 2.048 ms

    ; display ones for 2.048 ms

    decfsz   mpx_cnt,f        ; continue to multiplex display
    goto     mplex_loop      ; until 1 s has elapsed
```

Since there are three digits displayed in the loop, and each is displayed for 2 ms (approx.), the total time through the loop is 6 ms, so the number of iterations until 1 second has elapsed is $1\text{ s} \div 6\text{ ms} = 167$, small enough to fit into a single 8-bit counter, which is why a tick period of approximately 2 ms was chosen.

Note that, even if the internal RC oscillator was 100% accurate, giving an instruction clock of exactly 1 MHz, the time taken by this loop will be $162 \times 3 \times 2.048\text{ ms} = 995.3\text{ ms}$. Hence, this “clock” is guaranteed to be out by at least 0.5%. But accuracy isn’t the point of this exercise.

After displaying the current time for (close to) 1 second, we need to increment the time counters, and that can be done as follows:

```
; increment counters
    incf     ones,f           ; increment ones
    movlw   .10
    xorwf   ones,w           ; if ones overflow,
    btfss   STATUS,Z
    goto    end_inc
    clrf    ones             ; reset ones to 0
    incf    tens,f          ; and increment tens
    movlw   .6
    xorwf   tens,w          ; if tens overflow,
    btfss   STATUS,Z
    goto    end_inc
    clrf    tens            ; reset tens to 0
    incf    mins,f          ; and increment minutes
    movlw   .10
    xorwf   mins,w          ; if minutes overflow,
    btfsc   STATUS,Z
    clrf    mins            ; reset minutes to 0

end_inc
```

It’s simply a matter of incrementing the ‘ones’ digit as was done for a single digit, checking for overflows and incrementing the higher digits accordingly. The overflow (or *carry*) from seconds to minutes is done by testing for “tens = 6”. If you wanted to make this purely a seconds counter, counting from 0 to 999 seconds, you’d simply change this to test for “tens = 10”, instead.

After incrementing the time counters, the main loop begins again, displaying the updated time.

Complete program

Here is the complete program, incorporating the above code fragments.

One point to note is that **TMRO** is never initialised; there's no need, as it simply means that there may be a delay of up to 2 ms before the display begins for the first time, which isn't at all noticeable.

```

;*****
; Description: Lesson 8, example 2 *
; *
; Demonstrates use of multiplexing to drive multiple 7-seg displays *
; *
; 3 digit 7-segment LED display: 1 digit minutes, 2 digit seconds *
; counts in seconds 0:00 to 9:59 then repeats, *
; with timing derived from int 4 MHz oscillator *
; *
;*****
; Pin assignments: *
; RB0-1, RB4, RC1-4 = 7-segment display bus (common cathode) *
; RC5 = minutes enable (active high) *
; RB5 = tens enable *
; RC0 = ones enable *
; *
;*****

list p=16F506
#include <p16F506.inc>

radix dec

;***** CONFIGURATION
; ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG _MCLRE_ON & _CP_OFF & _WDT_OFF & _IOSCFS_OFF & _IntRC_OSC_RB4EN

; pin assignments
#define MINUTES PORTC,5 ; minutes enable
#define TENS PORTB,5 ; tens enable
#define ONES PORTC,0 ; ones enable

;***** VARIABLE DEFINITIONS
UDATA_SHR
temp res 1 ; used by set7seg routine (temp digit store)

UDATA
mpx_cnt res 1 ; multiplex counter
mins res 1 ; current count: minutes
tens res 1 ; tens
ones res 1 ; ones

;***** RC CALIBRATION
RCCAL CODE 0x3FF ; processor reset vector
res 1 ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET CODE 0x000 ; effective reset vector
movwf OSCCAL ; apply internal RC factory calibration
pagesel start
goto start ; jump to main code

```

```

;***** Subroutine vectors
set7seg          ; display digit on 7-segment display
    pagesel set7seg_R
    goto     set7seg_R

;***** MAIN PROGRAM *****
MAIN            CODE

;***** Initialisation
start
    ; configure ports
    clrw          ; configure PORTB and PORTC as all outputs
    tris    PORTB
    tris    PORTC
    clrf    ADCON0          ; disable AN0, AN1, AN2 inputs
    bcf     CM1CON0,C1ON    ; and comparator 1 -> RB0,RB1 digital
    bcf     CM2CON0,C2ON    ; disable comparator 2 -> RC0,RC1 digital

    ; configure timer
    movlw   b'11010111'    ; configure Timer0:
    ; --0-----          timer mode (T0CS = 0) -> RC5 usable
    ; ----0---          prescaler assigned to Timer0 (PSA = 0)
    ; -----111        prescale = 256 (PS = 111)
    option          ; -> increment every 256 us
    ;                ; (TMR0<2> cycles every 2.048ms)

    ; initialise variables
    banksel mins
    clrf    mins          ; start with count = 0:00
    clrf    tens
    clrf    ones

;***** Main loop
main_loop

; multiplex display for 1 sec
    movlw   1000000/2048/3 ; display each of 3 digits for 2.048 ms each
    movwf   mpx_cnt       ; repeat multiplex loop for approx 1 second

mplex_loop
    ; display minutes for 2.048 ms
w60_hi  btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto   w60_hi
        movf   mins,w          ; output minutes digit
        pagesel set7seg
        call   set7seg
        pagesel $
w60_lo  bsf     MINUTES          ; enable minutes display
        btfsc  TMR0,2          ; wait for TMR<2> to go low
        goto   w60_lo

    ; display tens for 2.048 ms
w10_hi  btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto   w10_hi
        movf   tens,w          ; output tens digit
        pagesel set7seg
        call   set7seg
        pagesel $
w10_lo  bsf     TENS            ; enable tens display
        btfsc  TMR0,2          ; wait for TMR<2> to go low
        goto   w10_lo

```

```

; display ones for 2.048 ms
w1_hi  btfss  TMR0,2      ; wait for TMR<2> to go high
        goto   w1_hi
        movf   ones,w    ; output ones digit
        pagesel set7seg
        call   set7seg
        pagesel $
w1_lo  btfsc  TMR0,2      ; wait for TMR<2> to go low
        goto   w1_lo

        decfsz mpx_cnt,f  ; continue to multiplex display
        goto   mplex_loop ; until 1 sec has elapsed

; increment time counters
        incf   ones,f    ; increment ones
        movlw .10
        xorwf  ones,w    ; if ones overflow,
        btfss STATUS,Z
        goto   end_inc
        clrf  ones      ; reset ones to 0
        incf  tens,f    ; and increment tens
        movlw .6
        xorwf  tens,w   ; if tens overflow,
        btfss STATUS,Z
        goto   end_inc
        clrf  tens     ; reset tens to 0
        incf  mins,f   ; and increment minutes
        movlw .10
        xorwf  mins,w   ; if minutes overflow,
        btfsc STATUS,Z
        clrf  mins     ; reset minutes to 0
end_inc

; repeat forever
        goto   main_loop

;***** LOOKUP TABLES *****
TABLES CODE 0x200 ; locate at beginning of a page

; pattern table for 7 segment display on port B
; RB4 = E, RB1:0 = FG
get7sB addwf PCL,f
        retlw b'010010' ; 0
        retlw b'000000' ; 1
        retlw b'010001' ; 2
        retlw b'000001' ; 3
        retlw b'000011' ; 4
        retlw b'000011' ; 5
        retlw b'010011' ; 6
        retlw b'000000' ; 7
        retlw b'010011' ; 8
        retlw b'000011' ; 9

; pattern table for 7 segment display on port C
; RC4:1 = CDBA
get7sC addwf PCL,f
        retlw b'011110' ; 0
        retlw b'010100' ; 1
        retlw b'001110' ; 2

```

```

    retlw    b'011110'      ; 3
    retlw    b'010100'      ; 4
    retlw    b'011010'      ; 5
    retlw    b'011010'      ; 6
    retlw    b'010110'      ; 7
    retlw    b'011110'      ; 8
    retlw    b'011110'      ; 9

; Display digit passed in W on 7-segment display
set7seg_R
    ; disable displays
    clrf    PORTB           ; clear all digit enable lines on PORTB
    clrf    PORTC           ; and PORTC

    ; output digit pattern
    movwf   temp           ; save digit
    call    get7sB         ; lookup pattern for port B
    movwf   PORTB          ; then output it
    movf    temp,w         ; get digit
    call    get7sC         ; then repeat for port C
    movwf   PORTC
    retlw   0

END

```

Binary-Coded Decimal

In the previous example, each digit in the time count was stored in its own 8-bit variable.

Since a single digit can only have values from 0 to 9, while an 8-bit register can store any integer from 0 to 255, it is apparent that storing each digit in a separate variable is an inefficient use of storage space. That can be an issue on devices with such a small amount of data memory – only 67 bytes on the 16F506.

The most space-efficient way to store integers is to use pure binary representation. E.g. the number ‘183’ would be stored in a single byte as b’10110111’ (or 0xB7). That’s three digits in a single byte. Of course, 3-digit numbers larger than 255 need two bytes, but any 4-digit number can be stored in two bytes, as can any 5-digit number less than 65536.

The problem with such “efficient” binary representation is that it’s difficult (i.e. time consuming) to unpack into decimal; necessary so that it can be displayed.

Consider how you would convert a number such as 0xB7 into decimal.

First, determine how many hundreds are in it. Baseline PICs do not have a “divide” instruction; the simplest approach is to subtract 100, check to see if there is a borrow, and subtract 100 again if there wasn’t (keeping track of the number of hundreds subtracted; this number of hundreds is the first digit):

$$0xB7 - 100 = 0x53$$

Now continue to subtract 10 from the remainder (0x53) until a borrow occurs, keeping track of how many tens were successfully subtracted, giving the second digit:

$$0x53 - (8 \times 10) = 0x03$$

The remainder (0x03) is of course the third digit.

Not only is this a complex routine, and takes a significant time to run (up to 12 subtractions are needed for a single conversion), it also requires storage; intermediate results such as “remainder” and “tens count” need to be stored somewhere.

Sometimes converting from pure binary into decimal is unavoidable, perhaps for example when dealing with quantities resulting from an analog to digital conversion (which we'll look at in [lesson 10](#)). But often, when storing numbers which will be displayed in decimal form, it makes sense to store them using *binary-coded decimal* representation.

In binary-coded decimal, or *BCD*, two digits are *packed* into each byte – one in each nybble (or “nibble”, as Microchip spells it).

For example, the BCD representation of 56 is 0x56. That is, each decimal digit corresponds directly to a hex digit when converted to BCD.

All eight bits in the byte are used, although not as efficiently as for binary. But BCD is far easier to work with for decimal operations, as we'll see.

Example application

To demonstrate the use of BCD, we'll modify the previous example to store “seconds” as a BCD variable.

So only two variables for the time count are now needed, instead of three:

```

                UDATA
mpx_cnt  res 1          ; multiplex counter
mins     res 1          ; time count: minutes
secs     res 1          ; seconds (BCD)

```

To display minutes is the same as before (since minutes is still being stored in its own variable), but to display the tens digit, we must first extract the digit from the high nybble, as follows:

```

                ; display tens for 2.048 ms
w10_hi  btfss  TMR0,2    ; wait for TMR0<2> to go high
        goto   w10_hi
        swapf  secs,w    ; get tens digit
        andlw  0x0F      ; from high nybble of seconds
        pagesel set7seg
        call   set7seg   ; then output it
        pagesel $

```

To move the contents of bits 4-7 (the high nybble) into bits 0-3 (the low nybble) of a register, you could use four ‘*rrf*’ instructions, to shift the contents of the register four bits to the right.

But the baseline PICs provide a very useful instruction for working with BCD: ‘*swapf f,d*’ – “**swap** nybbles in file register”. As usual, ‘*f*’ is the register supplying the value to be swapped, and ‘*d*’ is the destination: ‘*,f*’ to write the swapped result back to the register, or ‘*,w*’ to place the result in *W*.

Having gotten the tens digit into the lower nybble (in *W*, since we don't want to change the contents of the ‘*secs*’ variable), the upper nybble has to be cleared, so that only the tens digit is passed to the ‘*set7seg*’ routine.

This is done through a technique called *masking*.

It relies on the fact that any bit ANDed with ‘1’ remains unchanged, while any bit ANDed with ‘0’ is cleared to ‘0’. That is:

$$n \text{ AND } 1 = n$$

$$n \text{ AND } 0 = 0$$

So if a byte is ANDed with binary 00001111, the high nybble will be cleared, leaving the low nybble unchanged.

So far we've only seen the exclusive-or instructions, but the baseline PICs provide equivalent instructions for the logical "and" and "or" operations, including 'andlw', which ANDs a literal value with the contents of W, placing the result in W – "and literal with W".

So the 'andlw 0x0F' instruction masks off the high nybble, leaving only the tens digit left in W, to be passed to the 'set7seg' routine.

And why express the *bit mask* in hexadecimal (0x0F) instead of binary (b'00001111')? Simply because, when working with BCD values, hexadecimal notation seems clearer.

Extracting the ones digit is simply a masking operation, as the ones digit is already in the lower nybble:

```

; display ones for 2.048 ms
w1_hi  btfss   TMR0,2           ; wait for TMR0<2> to go high
        goto   w1_hi
        movf   secs,w          ; get ones digit
        andlw  0x0F           ; from low nybble of seconds
        pagesel set7seg
        call   set7seg        ; then output it
        pagesel $

```

The only other routine that has to be done differently, due to storing seconds in BCD format, is incrementing the time count, as follows:

```

; increment time counters
        incf   secs,f          ; increment seconds
        movf   secs,w          ; if ones overflow,
        andlw  0x0F           ;
        xorlw  .10
        btfss  STATUS,Z
        goto   end_inc
        movlw  .6              ; BCD adjust seconds
        addwf  secs,f
        movlw  0x60
        xorwf  secs,w          ; if seconds = 60,
        btfss  STATUS,Z
        goto   end_inc
        clrf   secs           ; reset seconds to 0
        incf   mins,f         ; and increment minutes
        movlw  .10
        xorwf  mins,w          ; if minutes overflow,
        btfsc  STATUS,Z
        clrf   mins           ; reset minutes to 0
end_inc

```

To check to see whether the 'ones' digit has been incremented past 9, it is extracted (by masking) and tested to see if it equals 10. If it does, then we need to reset the 'ones' digit to 0, and increment the 'tens' digit. But remember that BCD digits are essentially hexadecimal digits. The 'tens' digit is really counting by 16s, as far as the PIC is concerned, which operates purely on binary numbers, regardless of whether we consider them to be in BCD format. If the 'ones' digit is equal to 10, then adding 6 to it would take it to 16, which would overflow, leaving 'ones' cleared to 0, and incrementing 'tens'.

Putting it another way, you could say that adding 6 adjusts for BCD digit overflow. Some microprocessors provide a "decimal adjust" instruction, that performs this adjustment. The PIC doesn't, so we do it manually.

Finally, note that to check for seconds overflow, the test is not for "seconds = 60", but "seconds = 0x60", i.e. the value to be compared is expressed in hexadecimal, because seconds is stored in BCD format. Forgetting to express the seconds overflow test in hex would be an easy mistake to make...

The rest of the code is exactly the same as before, so won't be repeated here (although the source files for all the examples are of course available for download from www.gooligum.com.au).

Overall, the BCD version uses 104 words of program memory, and 4 bytes of data memory, compared with 102 words and 5 bytes for the un-packed version.

Is saving a single byte of data memory worth the additional complexity and two extra words of program memory? In this example, probably not. But in cases where you need to store more data, adding instructions to pack and extract that data can be well worth while. As with any trade-off, "it depends".

Conclusion

That completes our survey of digital I/O with the baseline PIC devices. More is possible, of course, but to go much further in digital I/O, it is better to make the jump to the midrange architecture.

But before doing so, we'll take a look at analog inputs, using comparators (in [lesson 9](#)) and analog to digital conversion (in [lesson 10](#)).

Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 9: Analog Comparators

We've seen how to respond to simple on/off digital signals, but we live in an “analog” world; many of the sensors you will want your PIC-based projects to respond to, such as temperature or light, have smoothly varying outputs whose magnitude represents the value being measured; they are *analog* outputs.

In many cases, you will want to treat the output of an analog sensor as though it was digital (i.e. on or off, high or low), without worrying about the exact value. It may be that a pulse, that does not meet the requirements for a “digital” input, has to be detected: for example, the output of a Hall-effect sensor attached to a rotating part. Or we may simply wish to respond to a threshold (perhaps temperature) being crossed.

In such cases, where we only need to respond to an input voltage being higher or lower than a threshold, it is usually appropriate to use an analog *comparator*. In fact, analog comparators are so useful that most PICs include them, as built-in peripherals.

This lesson explains how to use comparators, on baseline PIC devices¹, to respond to analog inputs.

In summary, this lesson covers:

- Using comparators to compare voltage levels
- Adding comparator hysteresis
- Using a comparator to drive Timer0
- Using absolute and programmable voltage references

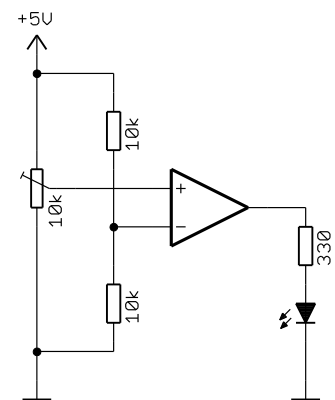
Comparators

A *comparator* (technically, an analog comparator, since comparators with digital inputs also exist) is a device which compares the voltages present on its positive and negative inputs. If the voltage on the positive input is greater than that on the negative input, the output is set “high”; otherwise the output is “low”.

An example is shown in the diagram on the right.

The two 10 kΩ resistors act as a voltage divider, presenting 2.5 V at the comparator's negative input. This sets the on/off threshold voltage.

The potentiometer can be adjusted to set the positive input to any voltage between 0 V and 5 V.



¹ The comparators on most baseline PICs (those with comparators), including the 10F2xx series, are very similar to those in the PIC16F506, used in this lesson.

When the potentiometer is set to a voltage under 2.5 V, the comparator's output will be low and the LED will not be lit. But when the potentiometer is turned up past halfway, the comparator's output will go high, lighting the LED.

Comparators are typically used when a circuit needs to react to a sensor's analog output being above or below some threshold, triggering some event (e.g. time to fill the tank, turn off a heater, or start an alarm).

They are also useful for level conversion. Suppose a sensor is outputting pulses which are logically "on/off" (i.e. the output is essentially digital), but do not match the voltage levels needed by the digital devices they are driving. For example, the digital inputs of a PIC, with $V_{DD} = 5\text{ V}$ (i.e. TTL-compatible), require at least 2.0 V to register as "high". That's a problem if a sensor is delivering a stream of 0 - 1 V pulses. By passing this signal through a comparator with an input threshold of 0.5 V, the 1 V pulses would be recognised as being "high".

Similarly, comparators can be used to *shape* or *condition* poorly defined or slowly-changing signals. The logic level of a signal between 0.8 V and 2.0 V is not defined for digital inputs on a PIC with $V_{DD} = 5\text{ V}$. And excessive current can flow when a digital input is at an intermediate value. Digital input signals which spend any significant amount of time in this intermediate range should be avoided. Such input signals can be cleaned up by passing them through a comparator; the output will have sharply-defined transitions between valid digital voltage levels.

The PIC16F506 includes two comparators, having different capabilities, as discussed in the following sections.

Comparator 1

Comparator 1 is the simpler of the two comparators.

It is controlled by the CM1CON0 register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
CM1CON0	C1OUT	$\overline{\text{C1OUTEN}}$	C1POL	$\overline{\text{C1T0CS}}$	C1ON	C1NREF	C1PREF	$\overline{\text{C1WU}}$

The comparator's inputs are determined by C1NREF and C1PREF.

C1PREF selects which pin will be used as the *positive reference* (or input):

C1PREF = 1 selects C1IN+

C1PREF = 0 selects C1IN-

[Note that, despite its name, the C1IN- pin can be used as the comparator's positive reference.]

C1NREF selects the *negative reference*:

C1NREF = 1 selects the C1IN- pin

C1NREF = 0 selects a 0.6 V internal reference voltage

By default (after a power-on reset), every bit of CM1CON0 is set to '1'.

This selects the C1IN+ pin as the positive reference, and C1IN- as the negative reference – the "normal" way to use the comparator.

Alternatively, the internal 0.6 V *absolute* reference can be selected as the negative reference, freeing up one I/O pin, with either C1IN+ or C1IN- providing the positive reference. Selecting C1IN+ as the positive reference is clearer, but occasionally it might make more sense to use C1IN-, perhaps because it simplifies your PCB layout, or you may be using comparator 1 for multiple measurements, alternately comparing C1IN- with 0.6 V, and then C1IN+ with C1IN-. For example, the 0.6 V reference could be used to solve the problem of detecting 0 - 1 V pulses, mentioned earlier.

The comparator's output appears as the C1OUT bit: it is normally set to '1' if and only if the positive reference voltage is higher than the negative reference voltage.

That's the normal situation, but the operation of the comparator can be inverted by clearing the C1POL output *polarity* bit:

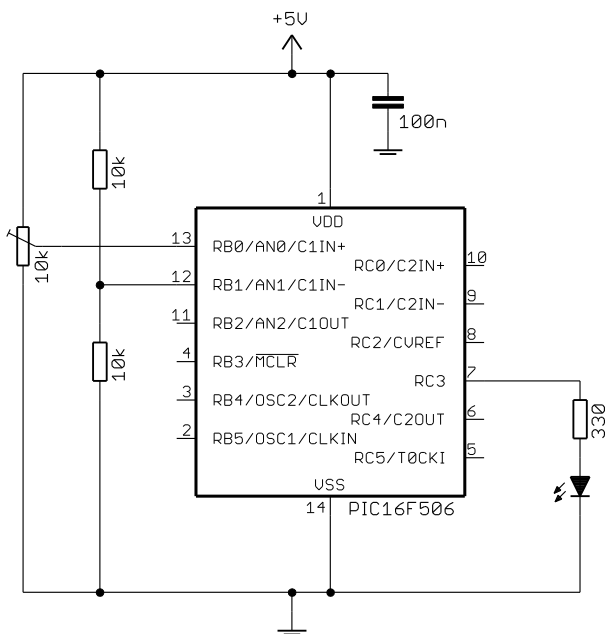
C1POL = 1 selects normal operation

C1POL = 0 selects inverted operation, where C1OUT = 1 only if positive ref < negative ref.

Finally, the C1ON bit turns the comparator on or off: '1' to turn it on, '0' to turn it off.

Those are the only bits needed for basic operation of comparator 1. We'll examine the other bits in the CM1CON1 register, later.

We will illustrate the comparator's basic operation using the circuit shown below.



If you have the [Gooligum baseline training board](#), you can implement this circuit by placing a shunt across pins 1 and 2 ('POT') of JP24, connecting the 10 kΩ pot (RP2) to C1IN+, and in JP19 to enable the LED on RC3.

The connection to C1IN- (labelled 'GP/RA/RB1' on the board) is available as pin 9 on the 16-pin header. +V and GND are brought out on pins 15 and 16, respectively, making it easy to add the 10 kΩ resistors (supplied with the board), forming a voltage divider, by using the solderless breadboard.

The circuit can alternatively be built using the Microchip Low Pin Count Demo Board.

The 10 kΩ potentiometer on that board is already connected to C1IN+ (labelled 'RA0' on the board), via a 1 kΩ resistor (not shown in this diagram). But you must ensure that jumper JP5 is closed; it will be,

if you haven't modified your demo board. The LED labelled 'DS4' on the demo board is connected to RC3 (via a 470 Ω resistor, instead of 330 Ω as shown here, but that makes no difference) via jumper JP4.

You can connect 10 kΩ resistors (which you will need to provide, along with a breadboard or other prototyping board) via the 14-pin header on the demo board. C1IN- (labelled 'RA1') is available on pin 8, and +V and GND are pins 13 and 14 respectively.

It is straightforward to configure the PIC as a simple comparator, turning on the LED if the potentiometer is turned more than halfway toward the +5V supply (right) side.

First configure RC3 as an output:

```
; configure ports
movlw  ~ (1<<nLED)      ; configure LED pin (only) as an output
tris   PORTC
```

There is no need to configure RB0 or RB1 as inputs, as the comparator settings override TRISB.

Next, we can configure comparator 1:

```
; configure comparator 1
movlw    1<<C1PREF|1<<C1NREF|1<<C1POL|1<<C1ON
        ; +ref is C1IN+ (C1PREF = 1)
        ; -ref is C1IN- (C1NREF = 1)
        ; normal polarity (C1POL = 1)
        ; comparator on (C1ON = 1)
movwf    CM1CON0
        ; -> C1OUT = 1 if C1IN+ > C1IN-
```

This turns comparator 1 on, and configures it to test for **C1IN+** > **C1IN-**. This is the default setting, so in theory these initialisation instructions are unnecessary. But relying implicitly on default settings is obscure and error-prone; it is much clearer to explicitly initialise the registers for the functions you are using.

To turn on the LED when the comparator output is high, repeatedly test **C1OUT**:

```
loop    btfsc    CM1CON0,C1OUT    ; if comparator output high
        bsf      LED              ; turn on LED
        btfss   CM1CON0,C1OUT    ; if comparator output low
        bcf      LED              ; turn off LED

        goto    loop              ; repeat forever
```

You should find that, as you turn the potentiometer past halfway, the LED turns on and off

To make the circuit a little more interesting, we'll add a light-dependent resistor (LDR, or CdS photocell), as shown on the right.

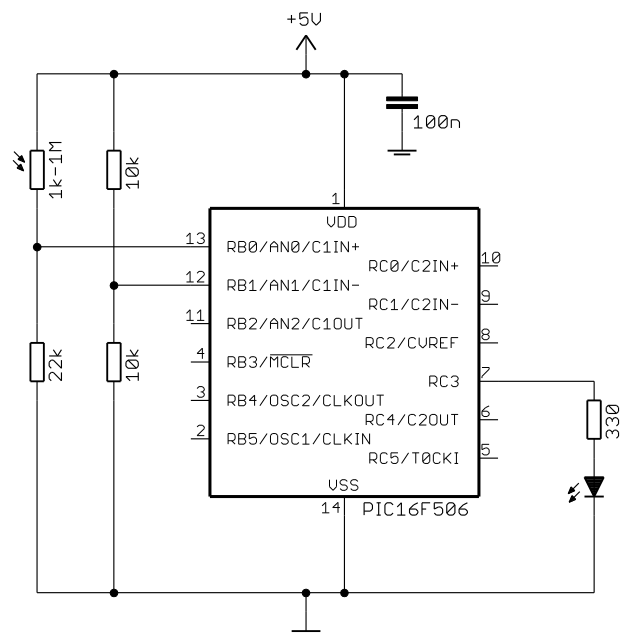
As the light level increases, the resistance of the LDR falls, and the voltage at the potential divider (formed by the LDR and the 22 kΩ resistor) connected to **C1IN+** rises.

The exact resistance range of the photocell is not important; the ones supplied with the [Gooligum baseline training board](#) have a resistance of around 20 kΩ or so for normal indoor lighting conditions, which is why a 22 kΩ resistor is used for the lower arm of the potential divider here – the voltage at **C1IN+** will vary around 2.5 V or so when it's not too bright, not too dark.

If you are using the [Gooligum baseline training board](#), you only need to move the shunt in JP24 across to pins 2 and 3 ('LDR1'), connecting the photocell (PH1) and 22 kΩ resistor in the lower left of the board to **C1IN+**.

If you are using the Microchip Low Pin Count Demo Board, you can't take the 10 kΩ potentiometer out of the circuit – it, and the 1 kΩ resistor in series between it and **C1IN+**, must be used as the "fixed" resistance, forming the lower arm of the potential divider. You must also remove jumper JP5 (you may need to cut the PCB trace – ideally you'd install a jumper, so that you can reconnect it again later), to disconnect the pot from the +5 V supply.

If you turn the pot all the way to the right, you'll have a total resistance of 11 kΩ between **C1IN+** and ground. That means that ideally you'd use a photocell with a resistance of around 10 kΩ with normal indoor lighting. The photocell can then be connected between pin 7 on the 14-pin header and +5 V.



If you don't have a photocell, there is no problem with continuing to use the potentiometer-only circuit for these lessons; but it's certainly more fun to build a circuit that responds to light!

When you build this circuit and use the above code, you will find that the LED turns on when the LDR is illuminated.

If you would prefer it to work the other way, so that the LED is lit when the light level falls (simulating e.g. the way that streetlamps turn on automatically when it gets dark), there's no need to change the circuit connections or program logic. Simply change the comparator initialisation instructions, to invert the output, by clearing the **C1POL** bit:

```

; configure comparator 1
movlw    1<<C1PREF|1<<C1NREF|0<<C1POL|1<<C1ON
                ; +ref is C1IN+ (C1PREF = 1)
                ; -ref is C1IN- (C1NREF = 1)
                ; inverted polarity (C1POL = 0)
                ; turn comparator on (C1ON = 1)
movwf    CM1CON0
                ; -> C1OUT = 1 if C1IN+ < C1IN-

```

We can save an I/O pin if we don't use an external voltage divider as the negative reference; we could instead use the 0.6 V internal reference, which is enabled by clearing the **C1NREF** bit:

```

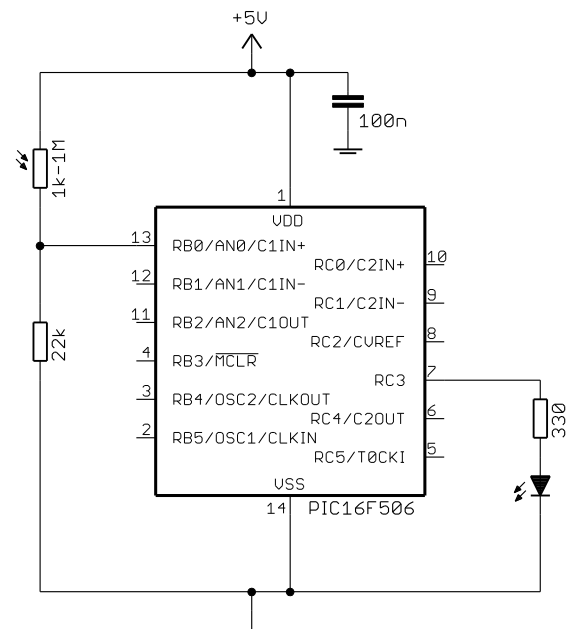
; configure comparator 1
movlw    1<<C1PREF|0<<C1NREF|0<<C1POL|1<<C1ON
                ; +ref is C1IN+ (C1PREF = 1)
                ; -ref is 0.6 V (C1NREF = 0)
                ; inverted polarity (C1POL = 0)
                ; turn comparator on (C1ON = 1)
movwf    CM1CON0
                ; -> C1OUT = 1 if C1IN+ < 0.6 V

```

Since the negative reference is now 0.6 V instead of 2.5 V, you will find that you will need to make it darker than before, to make the LED come on.

To convince yourself that it really is the internal reference voltage being used, you could remove the 10 kΩ voltage divider resistors, as shown on the right. You should find that removing the resistors makes no difference to the circuit's operation.

The important difference is that **RB1** is now available for use.



Adding hysteresis

You will notice, as the light level changes slowly past the threshold where the LED turns on and off, that the LED appears to fade in and out in brightness. This is caused by noise in the circuit and fluctuations in the light source (particularly at 50 or 60 Hz, from mains-powered incandescent or fluorescent lamps): when the input is very close to the threshold voltage, small input voltage variations due to noise and/or fluctuating

light levels cause the comparator to rapidly switch between high and low. This rapid switching, similar to switch bounce, can be a problem if the microcontroller is supposed to count input transitions, or to perform an action on each change.

To avoid this phenomenon, *hysteresis* can be added to a comparator, by adding positive feedback – feeding some of the comparator’s output back to its positive input.

Consider the comparator circuit shown on the right.

The threshold voltage, V_t , is set by a voltage divider, formed by resistors R_1 and R_2 .

This would normally set the threshold at $V_t = \frac{R_2}{R_1 + R_2} V_{dd}$.

However, resistor R_h feeds some of the comparator’s output back, increasing the threshold to some higher level, V_{th} , when the output is high, and decreasing it to a lower level, V_{tl} , when the output is low.

Now consider what happens when V_{in} is low (less than V_{tl}) and begins to increase. Initially, since $V_{in} < V_{tl}$, the comparator output is high, and the threshold is high: $V_t = V_{th}$.

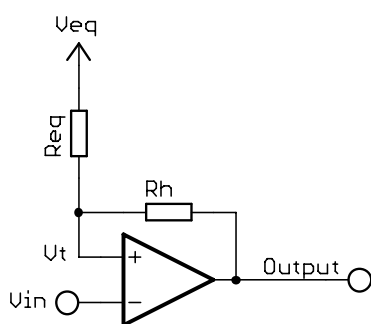
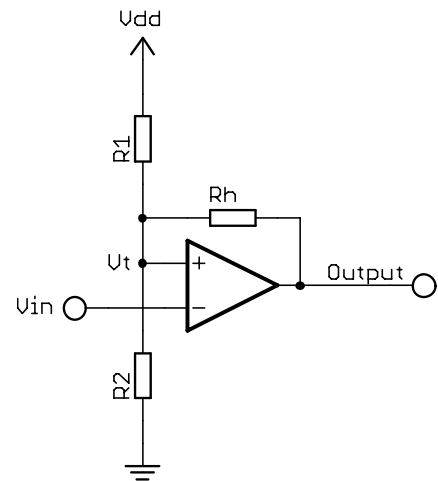
Eventually V_{in} rises above V_{th} , and the comparator output goes low, lowering the threshold: $V_t = V_{tl}$.

Now suppose the input voltage begins to fall. As it falls past the high threshold, V_{th} , nothing happens. It has to keep falling, all the way down to the low threshold, V_{tl} , before the comparator output changes again.

There are now two voltage thresholds: one (the higher) applying when the input signal is rising; the other (lower) applying when the input is falling. The comparator’s output depends not only on its inputs, but on their history – a key characteristic of hysteresis.

The voltage difference between the two thresholds is known as the hysteresis band: $V_{hb} = V_{th} - V_{tl}$.

This is the amount the input signal has to fall, after rising through the high threshold, or rise, after falling through the low threshold, before the comparator output switches again. It should be higher than the expected noise level in the circuit, making the comparator immune to most noise.



To calculate the high and low thresholds, recall Thévenin’s theorem, which states that any two-terminal network of resistors and voltage sources can be replaced by a single voltage source, V_{eq} , in series with a single resistor, R_{eq} .

Thus, the circuit above is equivalent to that on the left, where

$$V_{eq} = \frac{R_2}{R_1 + R_2} V_{dd}$$

and R_{eq} is the parallel combination of R_1 and R_2 : $R_{eq} = \frac{R_1 R_2}{R_1 + R_2}$

Therefore, when the comparator’s output is low ($= 0\text{ V}$):

$$V_{tl} = \frac{R_h}{R_h + R_{eq}} V_{eq} \quad (\text{thus } V_{tl} < V_{eq})$$

and, when the comparator's output is high (= V_{dd}):

$$V_{th} = V_{eq} + \frac{R_{eq}}{R_h + R_{eq}} (V_{dd} - V_{eq}) \quad (\text{thus } V_{th} > V_{eq})$$

This little bit of mathematics proves that $V_{th} > V_{tl}$, that is, the high input threshold is higher than the low input threshold.

The output of comparator 1 can be made available on the C1OUT pin (shared with RB2); we can use this to add hysteresis to the circuit.

The comparator output is enabled by clearing the $\overline{\text{C1OUTEN}}$ bit in the CM1CON0 register:

$\overline{\text{C1OUTEN}} = 0$ places the output of comparator 1 on the C1OUT pin

$\overline{\text{C1OUTEN}} = 1$ disables comparator output on C1OUT (i.e. normal operation).

Note: The comparator output overrides digital I/O. To use a pin for digital I/O, any comparator output assigned to that pin must be disabled. Comparator outputs are disabled on start-up.

In most examples of comparator hysteresis, the comparator's positive input is used as the threshold, and feedback is used to alter that threshold, as shown above.

However, in the example above, where the internal 0.6 V reference is used as the negative reference, it is not possible to use feedback to adjust the threshold; being an internal reference, there is no way to affect it.

But that's not a problem – it is also possible to introduce hysteresis by feeding the comparator output into the input signal (i.e. the signal being measured), assuming the input is connected to the positive input. It may not seem as intuitive, but the principle is essentially the same.

C1 on: C1IN+ < 0.6V When the input is higher than the threshold, the comparator output goes high, pulling the input even higher, via the feedback resistor. The circuit driving the input then has to effectively work harder, against the comparator's output, to bring the input back below the threshold. Similarly, when the input is low, the comparator's output goes low, dragging the input even lower, meaning that the input drive has to increase further before the comparator will change state again.

Suppose there is no feedback resistor and that the voltage on C1IN+ is equal to the threshold voltage of 0.6 V. This would happen if the light level is such that the LDR has a resistance of 161.3 k Ω :

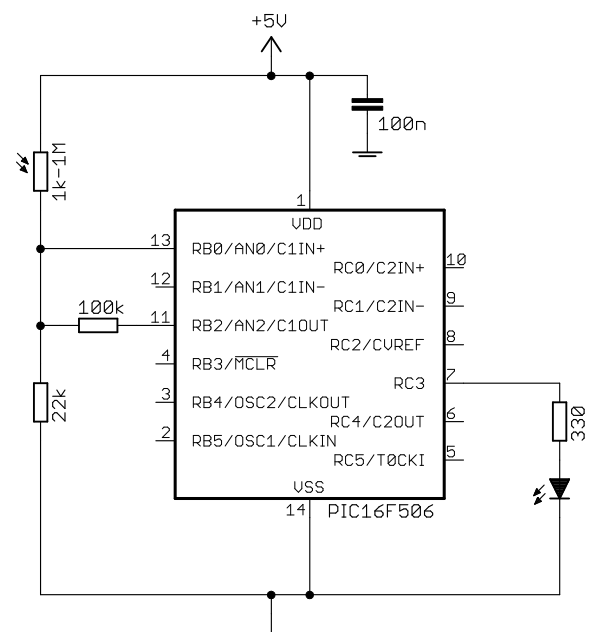
$$V_{in} = 22 / (22 + 161.3) \times 5 \text{ V} = 0.6 \text{ V}$$

With the input so close to the threshold, we would expect the output to jitter.

Now suppose that we add a 100 k Ω feedback resistor, as shown on the right.

You can do this with the [Gooligum baseline training board](#) by placing the supplied 100 k Ω resistor between pins 8 ('GP/RA/RB0') and 13 ('GP/RA/RB2') on the 16-pin header.

Or, if you are using the Microchip Low Pin Count Demo Board, you would place the feedback resistor between pins 7 and 9 on the 14-pin header.



The effect is similar to that for threshold voltage feedback. At that level of illumination, the potential divider formed by the LDR and the 22 kΩ resistor is equivalent to a source (input) voltage of 0.6 V, connected to C1IN+ via a resistance of:

$$R_{eq} = (161.3 \times 22) / (161.3 + 22) \text{ k}\Omega = 19.4 \text{ k}\Omega$$

When the comparator output is low, the feedback resistor pulls the input voltage down to:

$$V_{inl} = 100 / (100 + 19.4) \times 0.6 \text{ V} = 0.50 \text{ V}$$

When the comparator is high, the input voltage is pulled up to:

$$V_{inh} = 0.6 \text{ V} + 19.4 / (100 + 19.4) \times (5.0 \text{ V} - 0.6 \text{ V}) = 1.3 \text{ V}$$

Note that these voltages are not input thresholds. The comparator threshold is still the internal voltage reference of 0.6 V. These calculations only serve to demonstrate that the addition of positive feedback pulls the input lower when the comparator output is low and higher when the output is high, making the input less sensitive to small changes.

The code is essentially the same as before; since $\overline{\text{C1OUTEN}}$ must be cleared to enable the comparator output pin, that bit should not be set when CM1CON0 is loaded. Since the symbol for it ($\overline{\text{NOT_C1OUTEN}}$) hasn't been included in the examples so far, it has been cleared (through omission), so C1OUT has in fact always been enabled so far.

However, it is very important to set the C1POL bit. If it is not set, the comparator output, appearing on C1OUT, will be inverted, and the feedback will be negative, instead of the positive feedback needed to create hysteresis.

Instead, we can have the LED continue to indicate low light by inverting the display logic: light the LED only when C1OUT = 0.

But before the comparator output can actually appear on the C1OUT pin, that pin has to be deselected as an analog input.

Note: To enable comparator output on a pin, any analog input assigned to that pin must first be disabled.

Deselecting analog inputs is explained in the [next lesson](#) on analog-to-digital conversion, but for now all we need to remember is that all the analog inputs can be disabled by clearing the ADCON0 register.

The code for turning on an LED on RC3, when the photocell is not illuminated, using the internal 0.6 V reference, with hysteresis, is:

```

; configure ports
movlw  ~(1<<nLED)      ; configure LED pin (only) as an output
tris   PORTC
clrf   ADCON0         ; disable analog inputs -> C1OUT usable

; configure comparator 1
movlw  1<<<C1PREF|0<<<C1NREF|1<<<C1POL|0<<<NOT_C1OUTEN|1<<<C1ON
; +ref is C1IN+ (C1PREF = 1)
; -ref is 0.6 V (C1NREF = 0)
; normal polarity (C1POL = 1)
; enable C1OUT pin (/C1OUTEN = 0)
; turn comparator on (C1ON = 1)
movwf  CM1CON0       ; -> C1OUT = 1 if C1IN+ > 0.6V,
; C1OUT pin enabled

```

```

;***** Main loop
main_loop
    ; display comparator output (inverted)
    btfsc    CM1CON0,C1OUT    ; if comparator output high
    bcf      LED              ; turn off LED
    btfss    CM1CON0,C1OUT    ; if comparator output low
    bsf      LED              ; turn on LED

    ; repeat forever
    goto     main_loop

```

Wake-up on comparator change

As we saw in [lesson 7](#), most PICs can be put into standby, or sleep mode, to conserve power until they are woken by an external event. We've seen that that event can be a change on a digital input; it can also be a change on comparator output.

That's useful if your application is battery-powered and has to spend a long time waiting to respond to an input level change from a sensor.

Wake-up on change for comparator 1 is controlled by the $\overline{C1WU}$ bit in the CM1CON0 register.

By default (after a power-on reset), $\overline{C1WU} = 1$ and wake-up on comparator change is disabled.

To enable wake-up on comparator change, clear $\overline{C1WU}$.

Note: You should read the output of the comparator configured for wake-up on change just prior to entering sleep mode. Otherwise, if the comparator output had changed since the last time it was read, a "wake up on comparator change" reset will occur immediately upon entering sleep mode.

Since the symbol for setting $\overline{C1WU}$ ($\text{NOT } \overline{C1WU}$) has not been included in the expression loaded into CM1CON0 in any of the examples so far, $\overline{C1WU}$ has always been cleared (by omission), and wake-up on comparator change has been implicitly enabled every time.

But to explicitly enable wake-up on comparator change, you should use an expression such as:

```

movlw    1<<C1PREF|1<<C1NREF|0<<C1POL|0<<NOT_C1WU|1<<C1ON
movwf    CM1CON0

```

To determine whether a reset was due to a comparator change, test the CWUF flag in the STATUS register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	RBWUF	CWUF	PA0	\overline{TO}	\overline{PD}	Z	DC	C

CWUF = 1 indicates that a wake-up from sleep on comparator change reset occurred

CWUF = 0 after all other resets.

Although there are two comparators in the 16F506, this flag doesn't tell you which comparator's output changed; by default (after a power-on reset) wake-up on change for each comparator is disabled, but if you have enabled wake-up on change for both, you need to have stored the previous value of each comparator's output so that you can test to see which one changed. It's the same situation as for wake-up on change for the digital inputs; there is only one flag (GPWUF/RBWUF), but potentially a number of inputs that may have changed to trigger the wake-up. The only way to know what has changed is to compare each comparator output against the previously-recorded value.

The following example program configures Comparator 1 for wake-up on change and then goes to sleep. When the comparator output changes, the PIC resets, the wake-up on change condition is detected, and a LED is turned on for one second, to indicate that the comparator output has changed. The PIC then goes back to sleep, to wait until the next comparator output change.

We'll keep the same circuit as before, with the feedback resistor providing hysteresis in place, to make the comparator less sensitive to small variations – we only want the PIC to wake on a significant change, as you move the photocell between light to dark and back again.

Note that, when the comparator is initialised, there is a delay of 10 ms to allow it to settle before entering standby. This is necessary because signal levels can take a while to settle after power-on, and if the comparator output was changing while going into sleep mode, the PIC would immediately wake and the LED would flash – a false trigger. Note also that the comparator output is read (any instruction which reads CM1CON0 will do) immediately before the `sleep` instruction is executed, as explained above.

The delays are generated by the `DelayMS` macro, introduced in [lesson 6](#).

```
start
; configure ports
clrf    PORTC           ; start with LED off
movlw  ~(1<<nLED)      ; configure LED pin (only) as an output
tris   PORTC
clrf   ADCON0         ; disable analog inputs -> C1OUT usable

; check for wake-up on comparator change
btfsc  STATUS,CWUF    ; if wake-up on comparator change occurred,
goto   flash         ; flash LED then sleep

; else power-on reset
movlw  b'00111010'    ; configure comparator 1:
; -0-----          enable C1OUT pin (/C1OUTEN = 0)
; --1-----          normal polarity (C1POL = 1)
; ----1---           turn comparator on (C1ON = 1)
; -----0--         -ref is 0.6 V (C1NREF = 0)
; -----1-          +ref is C1IN+ (C1PREF = 1)
; -----0           enable wake on comparator change (/C1WU = 0)
movwf  CM1CON0        ; -> C1OUT = 1 if C1IN+ > 0.6V,
; C1OUT pin enabled,
; wake on comparator change enabled

DelayMS 10            ; delay 10 ms to allow comparator to settle

goto   standby       ; sleep until comparator change

;***** Main code
; flash LED
flash  bsf    LED      ; turn on LED
      DelayMS 1000    ; delay 1 sec

; sleep until comparator change
standby bcf    LED      ; turn off LED
      movf   CM1CON0,w  ; read comparator to clear mismatch condition
      sleep  ; enter sleep mode
```

Note that, in this example, the binary value 'b'01111010' was used to initialise the comparator control register, instead of the equivalent (but much longer!) expression:

```
'1<<C1PREF|0<<C1NREF|1<<C1POL|1<<NOT_C1OUTEN|0<<NOT_C1WU|1<<C1ON'
```

Either style is ok, as long as it is clearly commented.

Finally, you should be aware that, if a comparator is turned on when the PIC enters sleep mode, it will continue to draw current. If the comparator is to wake the PIC up when an input level changes, then of course it has to remain active, using power, while the PIC is in standby. But if you're not using wait on comparator change, you should turn off all comparators (clear **C1ON** to turn off comparator 1) before entering sleep mode, to save power.

Note: To minimise power consumption, turn comparators off before entering sleep mode.

Incrementing Timer0

We saw in [lesson 5](#) that Timer0 can be used as a counter, clocked by an external signal on **T0CKI**.

That's useful, but what if you want to count pulses that are not clean digital signals? The obvious answer is to pass the pulses through a comparator (with hysteresis, if the signal is noisy), and then feed the output of the comparator into **T0CKI**. Indeed, on some PICs, you need to make an electrical connection, external to the PIC, from the comparator output (e.g. the **C1OUT** pin) to the counter input (e.g. **T0CKI**).

However, on the 16F506, the output of comparator 1 can drive Timer0 directly, through an internal connection.

To enable the connection from comparator 1 to Timer0, clear the $\overline{\text{C1T0CS}}$ bit in the **CM1CON0** register:

$\overline{\text{C1T0CS}} = 1$ selects **T0CKI** as the Timer0 counter clock source

$\overline{\text{C1T0CS}} = 0$ selects the output of comparator 1 as the Timer0 counter clock source

Note that this setting only matters if Timer0 is in counter mode, i.e. **OPTION:T0CS** = 1.

If Timer0 is in timer mode (**T0CS** = 0), the timer will be incremented by the instruction clock (**FOSC/4**), regardless of the setting of $\overline{\text{C1T0CS}}$.

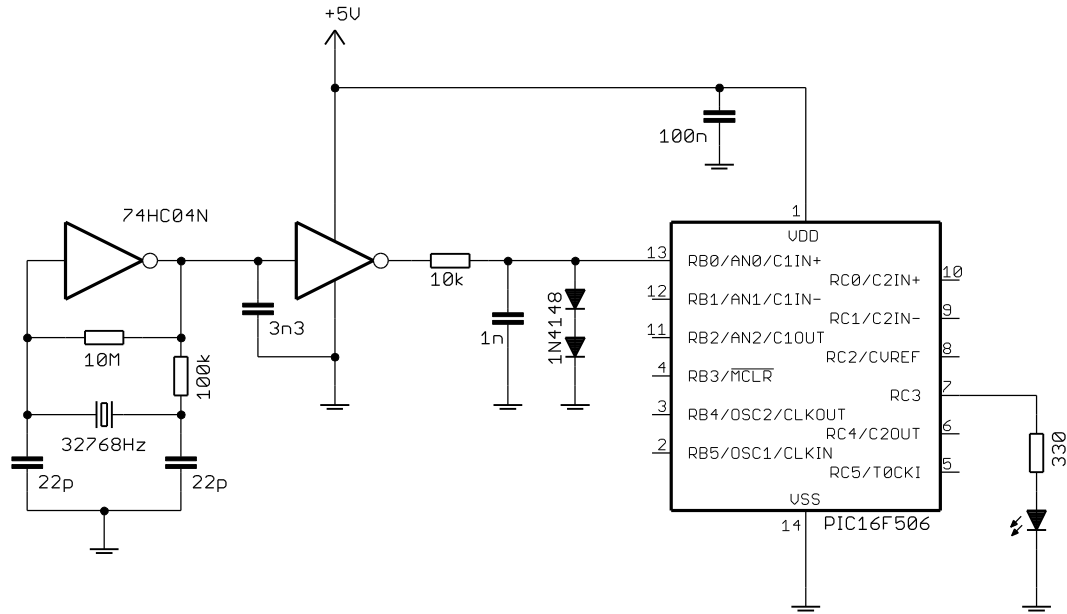
So, to use comparator 1 as the counter clock source, you must set **T0CS** = 1 and $\overline{\text{C1T0CS}} = 0$.

In this mode, Timer0 otherwise operates as usual, with the **T0SE** bit determining whether the counter increments on the rising or falling edge of the comparator output, and the **PSA** and **PS<2:0>** bits selecting the prescaler assignment.

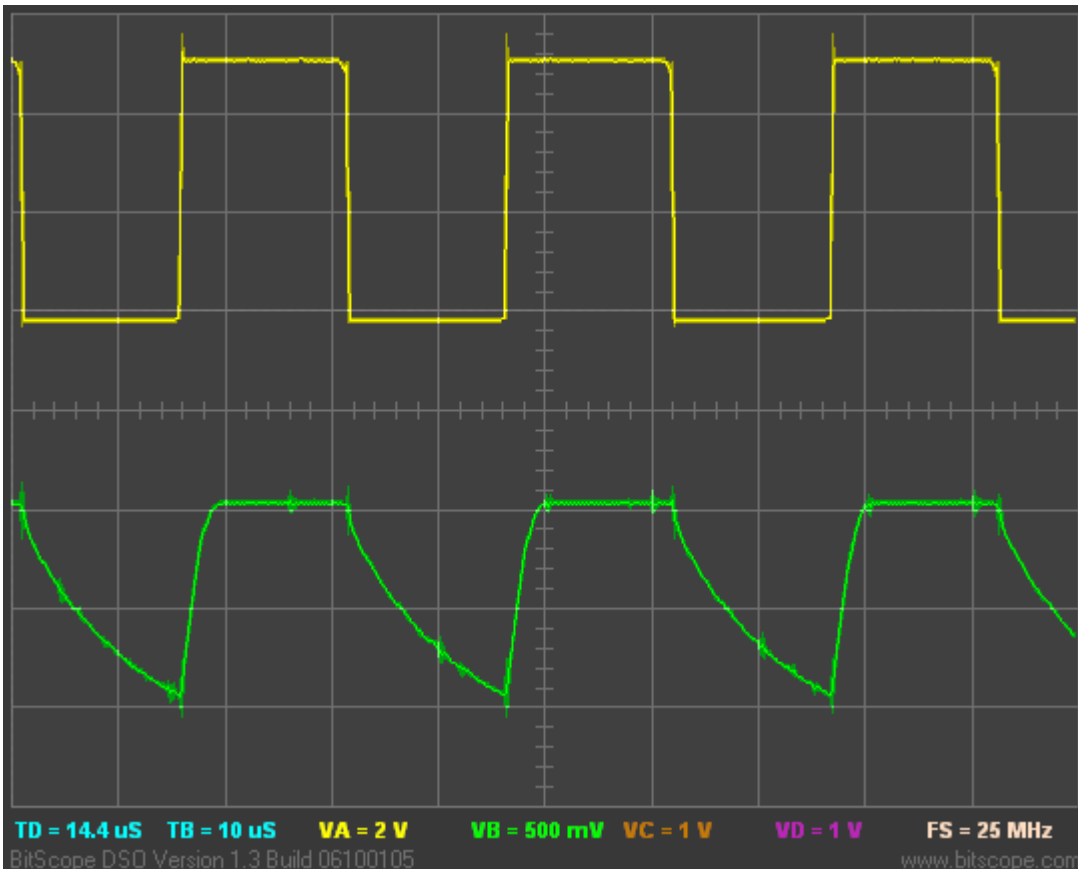
There is one quirk to be aware of: when **T0CS** is set to '1', the **TRIS** setting for **RC5** is overridden, and **RC5** cannot be used as a digital output – regardless of whether **T0CKI** is actually used as the counter clock source.

Note: When using comparator 1 to drive Timer0, RC5 cannot be used as an output – even though the T0CKI function on that pin is not being used.

To show how comparator 1 can be used with Timer0, we can use the external clock circuit from [lesson 5](#), but with the clock signal degraded by passing it through an RC filter and clamping with two diodes, as shown:

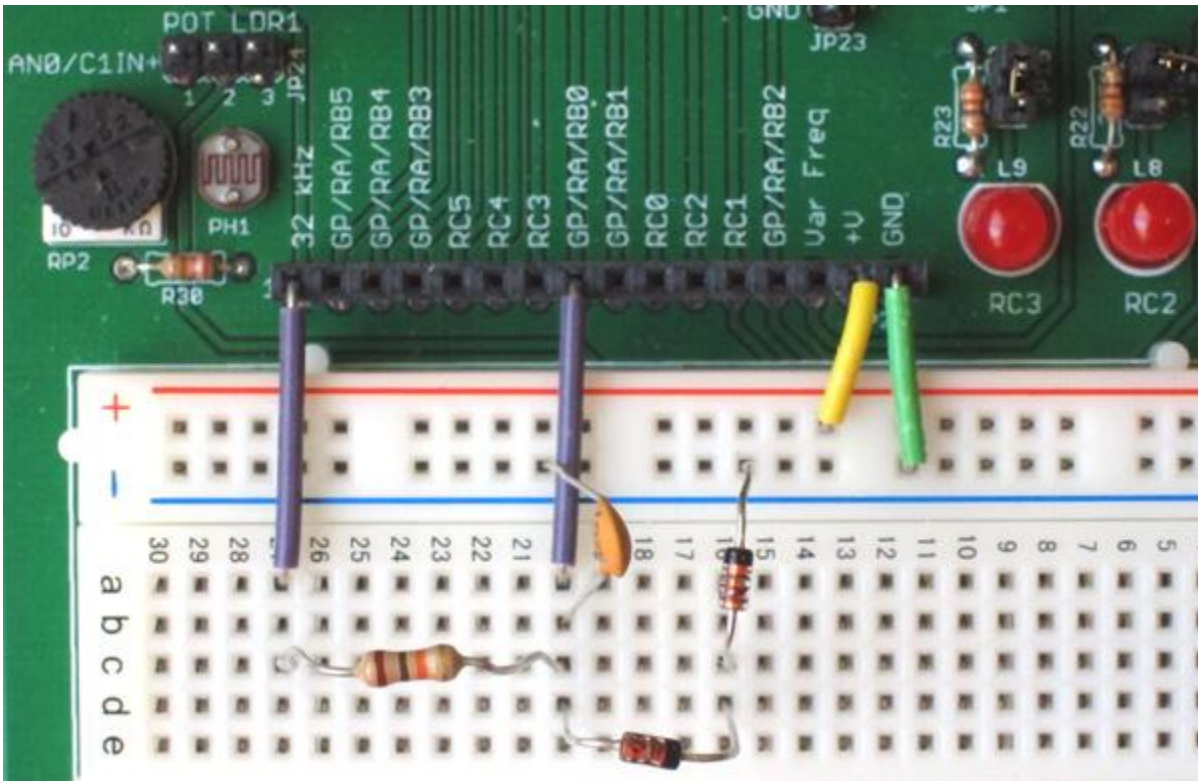


The effect of this can be seen in the oscilloscope trace, below:



The top trace is the “clean” digital output of the clock circuit, and the degraded signal is below. It peaks at approximately 1 V.

You can build this circuit with the [Gooligum baseline training board](#), using the supplied 10 kΩ resistor, 1 nF capacitor and two 1N4148 diodes – connecting them to signals on the 16-pin header: 32.768 kHz oscillator output on pin 1 (‘32 kHz’), C1IN+ input on pin 8 (‘GP/RA/RB0’) and ground on pin 16 (‘GND’) – using the solderless breadboard, as illustrated below:



You should also remove the shunt from JP24 (disconnecting the pot or photocell from C1IN+), but leave JP19 in place (enabling the LED on RC3).

If you are using Microchip’s Low Pin Count Demo Board, you can build the circuit in a similar way, by making connections to the 14-pin header on that board, although you will of course have to supply your own 32.768 kHz oscillator. *Note that, if you are using Microchip’s LPC Demo Board, the components connected to the C1IN+ input, including the potentiometer, will affect the signal on C1IN+. You may need to adjust the pot to make it work.*

Whichever board you use, you must only connect these additional components to C1IN+ **after** programming the PIC, to avoid interference with the programming process. You need to program the PIC before making the connection to C1IN+. You can then apply power (whether from a PICKit 2, PICKit3, or external power supply) and release reset – and the LED on RC3 should start flashing.

Note: Components such as diodes and capacitors connected to the ICSP programming pins (RB0/AN0/C1IN+ and RB1/AN1/C1IN- on a PIC16F506) may interfere with the ICSP programming signals, and must be disconnected before the PIC can be successfully programmed.

The degraded signal is not suitable for driving T0CKI directly; it doesn’t go high enough to register as a digital “high”. But it is quite suitable for passing through a comparator referenced to 0.6 V, so the internal reference voltage is a good choice here. Therefore the only comparator input needed is C1IN+, as shown in the circuit on the previous page.

[Lesson 5](#) included a program which flashed an LED at 1 Hz, with Timer0 driven by an external 32.768 kHz clock. It needs very little modification to work with comparator 1 instead of T0CKI. Other than changing references from GPIO to PORTC (since we are now using a 16F506 instead of a 12F509), all we need do is to configure the comparator, with comparator output selected as the Timer0 clock source:

```

; configure comparator 1
movlw    1<<C1PREF|0<<C1NREF|0<<C1POL|0<<NOT_C1T0CS|1<<C1ON
        ; +ref is C1IN+ (C1PREF = 1)
        ; -ref is 0.6 V (C1NREF = 0)
        ; normal polarity (C1POL = 1)
        ; select C1 as TMR0 clock (/C1T0CS = 0)
        ; turn comparator on (C1ON = 1)
movwf    CM1CON0
        ; -> C1OUT = 1 if C1IN+ > 0.6V,
        ;      TMR0 clock from C1

```

Once again, although it is not strictly necessary to include '0<<NOT_C1T0CS' in the expression being loaded into CM1CON0 (since ORing a zero value into an expression has no effect), doing so makes it explicit that we are enabling comparator 1 as a clock source, by clearing C1T0CS.

Complete program

Here is how the example from lesson 5 (actually the [lesson 6](#) version) has been modified:

```

;*****
;
; Description:      Lesson 9, example 3
;                  Crystal-based (degraded signal) LED flasher
;
; Demonstrates comparator 1 clocking TMR0
;
; LED flashes at 1 Hz (50% duty cycle),
; with timing derived from 32.768 kHz input on C1IN+
;
;*****
;
; Pin assignments:
;   C1IN+ = 32.768 kHz signal
;   RC3   = flashing LED
;
;*****

list      p=16F506
#include  <p16F506.inc>

radix    dec

;***** CONFIGURATION
        ; ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG _MCLRE_ON & _CP_OFF & _WDT_OFF & _IOSCFSS_OFF & _IntRC_OSC_RB4EN

; pin assignments
constant nFLASH=3                ; flashing LED on RC3

;***** VARIABLE DEFINITIONS
        UDATA_SHR
sPORTC  res 1                    ; shadow copy of PORTC

```

```

;***** RC CALIBRATION
RCCAL   CODE    0x3FF           ; processor reset vector
        res 1                   ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET   CODE    0x000           ; effective reset vector
        movwf OSCCAL            ; apply internal RC factory calibration

;***** MAIN PROGRAM *****

;***** Initialisation
start
        ; configure ports
        movlw  ~(1<<nFLASH)      ; configure LED pin (only) as output
        tris   PORTC

        ; configure Timer0
        movlw  1<<T0CS|0<<PSA|b'110'
                                   ; counter mode (T0CS = 1)
                                   ; prescaler assigned to Timer0 (PSA = 0)
                                   ; prescale = 128 (PS = 110)
        option                                ; -> incr at 256 Hz with 32.768 kHz input

        ; configure comparator 1
        movlw  1<<C1PREF|0<<C1NREF|0<<C1POL|0<<NOT_C1T0CS|1<<C1ON
                                   ; +ref is C1IN+ (C1PREF = 1)
                                   ; -ref is 0.6 V (C1NREF = 0)
                                   ; normal polarity (C1POL = 1)
                                   ; select C1 as TMR0 clock (/C1T0CS = 0)
                                   ; turn comparator on (C1ON = 1)
        movwf  CM1CON0            ; -> C1OUT = 1 if C1IN+ > 0.6V,
                                   ; TMR0 clock from C1

;***** Main loop
main_loop
        ; TMR0<7> cycles at 1 Hz, so continually copy to LED (GP1)
        clrf  sPORTC             ; assume TMR0<7>=0 -> LED off
        btfsc TMR0,7            ; if TMR0<7>=1
        bsf   sPORTC,nFLASH     ; turn on LED

        movf  sPORTC,w          ; copy shadow to port
        movwf PORTC

        ; repeat forever
        goto  main_loop

        END

```

Comparator 2

Comparator 2 is quite similar to comparator 1, but a wider range of inputs can be selected and it can be used with the programmable voltage reference.

It cannot, however, be selected as a clock source for Timer0.

It is controlled by the CM2CON0 register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
CM2CON0	C2OUT	$\overline{\text{C2OUTEN}}$	C2POL	C2PREF2	C2ON	C2NREF	C2PREF1	$\overline{\text{C2WU}}$

Most of these bits are directly equivalent to those in CM1CON0:

- C2OUT is the comparator output bit (the output appears here)
- $\overline{\text{C2OUTEN}}$ determines whether the output is placed on the C2OUT pin or not
- C2POL selects the output polarity
- C2ON turns the comparator on or off
- $\overline{\text{C2WU}}$ enables/disables wake-up on comparator change

They have the same options and work in the same way as the corresponding bits in CM1CON0.

The C2PREF bits select the positive reference:

C2PREF1	C2PREF2	Positive Reference
0	0	C2IN-
0	1	C1IN+
1	-	C2IN+

Note that C1IN+ (a “comparator 1” input) can be selected as an input to comparator 2.

Furthermore, C1IN+ can be used as an input to both comparators at the same time – something that can be useful if you want to compare two signals (one for each comparator) against a common external (perhaps varying) reference

on C1IN+. Or, the two comparators could be used to define upper and lower limits for the signal on C1IN+.

C2NREF selects the negative reference:

- C2NREF = 1 selects the C2IN- pin
- C2NREF = 0 selects the programmable internal reference voltage (see below)

By default (after a power-on reset), every bit of CM2CON0 is set to ‘1’.

Programmable voltage reference

We’ve seen that comparator 1 can be used with a fixed 0.6 V internal voltage reference, avoiding the need to provide an external reference voltage and saving a pin. However, 0.6 V is not always suitable, so an external reference may need to be used.

Comparator 2 is more flexible, in that it can be used with a programmable internal voltage reference (CVREF), selectable from a range of 32 voltages, from 0V to $0.72 \times VDD$.

The voltage reference is controlled by the VRCON register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
VRCON	VREN	VROE	VRR	-	VR3	VR2	VR1	VR0

The reference voltage is set by the VR<3:0> bits and VRR, which selects a high or low voltage range:

- VRR = 1 selects the low range, where $CVREF = VR<3:0>/24 \times VDD$.
- VRR = 0 selects the high range, where $CVREF = VDD/4 + VR<3:0>/32 \times VDD$.

The available reference voltages are summarised in the following table, as a fraction of V_{DD} and as an absolute voltage for the case where $V_{DD} = 5\text{ V}$:

VRR = 1 (low range)		
VR<3:0>	fraction V_{DD}	CVREF ($V_{DD} = 5\text{V}$)
0	0.000	0.00V
1	0.042	0.21V
2	0.083	0.42V
3	0.125	0.62V
4	0.167	0.83V
5	0.208	1.04V
6	0.250	1.25V
7	0.292	1.46V
8	0.333	1.67V
9	0.375	1.87V
10	0.417	2.08V
11	0.458	2.29V
12	0.500	2.50V
13	0.542	2.71V
14	0.583	2.92V
15	0.625	3.12V

VRR = 0 (high range)		
VR<3:0>	fraction V_{DD}	CVREF ($V_{DD} = 5\text{V}$)
0	0.250	1.25V
1	0.281	1.41V
2	0.313	1.56V
3	0.344	1.72V
4	0.375	1.88V
5	0.406	2.03V
6	0.438	2.19V
7	0.469	2.34V
8	0.500	2.50V
9	0.531	2.66V
10	0.563	2.81V
11	0.594	2.97V
12	0.625	3.13V
13	0.656	3.28V
14	0.688	3.44V
15	0.719	3.59V

Note that the low and high ranges overlap, with $0.250 \times V_{DD}$, $0.500 \times V_{DD}$ and $0.625 \times V_{DD}$ selectable in both. Thus, of the 32 selectable voltages, only 29 are unique.

The VREN bit enables (turns on) the voltage reference.

To use the voltage reference, set VREN = 1.

Since the voltage reference module draws current, you should turn it off by clearing VREN to minimise power consumption in sleep mode – unless of course you are using wake on comparator change with CVREF as the negative reference, in which case the voltage reference needs to remain on.

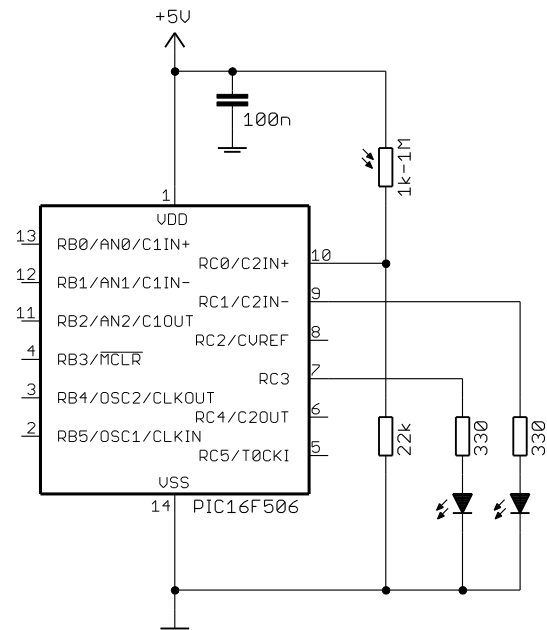
The only exception to this is you wish to set $CVREF = 0\text{V}$. In that case, with VRR = 1 and VR<3:0> = 0, the module can be turned off (VREN = 0) to conserve power and the reference voltage will be very close to 0V. That's useful if you wish to test for the input signal going negative (*zero-crossing*); the inputs can accept small negative voltages, down to -0.3V.

The VROE bit enables the voltage reference output on the CVREF pin. When VROE = 1, the CVREF output is enabled and, since it shares a pin with RC2, RC2 cannot then be used for digital I/O.

To demonstrate how the programmable voltage reference can be used with comparator 2, we can use the circuit shown on the right, with a photocell (and 22 kΩ resistor) connected to C2IN+. The LED on RC3 will indicate a low level of illumination, and the LED on RC1 will indicate bright light. When neither LED is lit, the light level will be in the middle; not too dim or too bright.

To implement this circuit using the [Gooligum baseline training board](#), you should remove the shunt from of JP24 and instead place a shunt in position 2 ('C2IN+') of JP25, connecting photocell PH2 to C2IN+. You should also place shunts in JP17 and JP19, enabling the LEDs on RC1 and RC3.

If you are using Microchip's Low Pin Count Demo Board, you can connect a photocell and resistor (or pot) to C2IN+ via pin 10 on the 14-pin header. The demo board already has LEDs on RC1 and RC3.



To test whether the input is within limits, we will first configure the programmable voltage reference to generate the “low” threshold voltage, compare the input with this low level, and then reconfigure the voltage reference to generate the “high” threshold and compare the input with this higher level.

This process could be extended to multiple input thresholds, by configuring the voltage reference to generate each threshold in turn. However, if you wish to test against more than a few threshold levels, you would probably be better off using an analog-to-digital converter (described in the [next lesson](#)).

This example uses 2.0 V as the “low” threshold and 3.0 V as the “high” threshold, but, since the reference is programmable, you can always choose your own levels!

Comparator 2 is configured to use C2IN+ as the positive reference, and CVREF as the negative reference:

```
; configure comparator 2
movlw    1<<C2PREF1|0<<C2NREF|1<<C2POL|1<<C2ON
; +ref is C2IN+ (C2PREF1 = 1)
; -ref is CVref (C2NREF = 0)
; normal polarity (C2POL = 1)
; turn comparator on (C2ON = 1)
movwf    CM2CON0
; -> C2OUT = 1 if C2IN+ > CVref
```

The voltage reference can be configured to generate approximately 2.0 V, by:

```
movlw    1<<VREN|0<<VRR|.5    ; configure voltage reference:
; enable voltage reference (VREN = 1)
; CVref = 0.406*Vdd (VRR = 0, VR = 5)
movwf    VRCON
; -> CVref = 2.03 V
```

The closest match to 3.0 V is obtained by:

```
movlw    1<<VREN|0<<VRR|.11   ; configure voltage reference:
; enable voltage reference (VREN = 1)
; CVref = 0.594*Vdd (VRR = 0, VR = 11)
movwf    VRCON
; -> CVref = 2.97 V
```

After changing the voltage reference, it can take a little while for it to settle and stably generate the newly-selected voltage. According to the PIC12F510/16F506 data sheet, this settling time can be up to 10 μ s.

Therefore, we should insert a 10 μ s delay after configuring the voltage reference, before reading the comparator output.

As we saw in [lesson 2](#), a useful instruction for generating a two-instruction-cycle delay is 'goto \$+1'. Since each instruction cycle is 1 μ s (with a 4 MHz processor clock), each 'goto \$+1' creates a 2 μ s delay, and five of these instructions will give us the 10 μ s delay we are after.

Since we need to insert this delay twice (once for each time we re-configure the voltage reference), it makes sense to define it as a macro:

```
; 10 us delay
; (assuming 4 MHz processor clock)
Delay10us    MACRO
               goto $+1           ; 2 us delay * 5 = 10 us
               goto $+1
               goto $+1
               goto $+1
               goto $+1
               ENDM
```

This 'Delay10us' macro can then be used to add the necessary 10 μ s delay after each voltage reference re-configuration.

Complete program

Here is how these code fragments fit together.

Note that a shadow register is used for PORTC – in this case not so much to avoid read-write-modify problems, but simply because it makes the main loop logic more straightforward. If the loop started with PORTC being cleared, and then one of the LEDs turned on, the LED would end up being turned off then on, rapidly. The flickering would be too fast to be visible, but the LED's apparent brightness would be lower. Using a shadow register avoids that, without having to add more complex logic.

```
*****
; Description:      Lesson 9, example 4                                     *
;                                                         *
; Demonstrates use of Comparator 2 and programmable voltage reference *
;                                                         *
; Turns on Low LED  when C2IN+ < 2.0 V (low light level)           *
;                   or High LED when C2IN+ > 3.0 V (high light level) *
;                                                         *
;*****
;
; Pin assignments:                                               *
;   C2IN+ = voltage to be measured (LDR/resistor divider)         *
;   RC3   = "Low" LED                                             *
;   RC1   = "High" LED                                           *
;                                                         *
;*****

list          p=16F506
#include      <p16F506.inc>

radix        dec

;***** CONFIGURATION
               ; ext reset, no code protect, no watchdog, 4 Mhz int clock
__CONFIG     _MCLRE_ON & _CP_OFF & _WDT_OFF & _IOSCFSS_OFF & _IntRC_OSC_RB4EN
```

```

; pin assignments
    constant    nLO=RC3          ; "Low" LED
    constant    nHI=RC1          ; "High" LED

;***** MACROS
; 10 us delay
; Assumes: 4 MHz processor clock
;
Delay10us    MACRO
    goto $+1          ; 2 us delay * 5 = 10 us
    goto $+1
    goto $+1
    goto $+1
    goto $+1
    ENDM

;***** VARIABLE DEFINITIONS
    UDATA_SHR
sPORTC    res 1          ; shadow copy of PORTC

;***** RC CALIBRATION
RCCAL    CODE    0x3FF          ; processor reset vector
    res 1          ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET    CODE    0x000          ; effective reset vector
    movwf    OSCCAL          ; apply internal RC factory calibration

;***** MAIN PROGRAM *****

;***** Initialisation
start
    ; configure ports
    movlw    ~(1<<nLO|1<<nHI)    ; configure PORTC LED pins as outputs
    tris    PORTC

    ; configure comparator 2
    movlw    1<<<C2PREF1|0<<<C2NREF|1<<<C2POL|1<<<C2ON
    ; +ref is C2IN+ (C2PREF1 = 1)
    ; -ref is CVref (C2NREF = 0)
    ; normal polarity (C2POL = 1)
    ; turn comparator on (C2ON = 1)
    movwf    CM2CON0          ; -> C2OUT = 1 if C2IN+ > CVref

;***** Main loop
main_loop
    ; start with shadow PORTC clear
    clrf    sPORTC

;*** Test for low illumination
    ; set low input threshold
    movlw    1<<<VREN|0<<<VRR|.5    ; configure voltage reference:
    ; enable voltage reference (VREN = 1)
    ; CVref = 0.406*Vdd (VRR = 0, VR = 5)
    movwf    VRCON          ; -> CVref = 2.03 V
    Delay10us          ; wait 10 us to settle

```

```

; compare with input
btfss CM2CON0,C2OUT ; if C2IN+ < CVref
bsf sPORTC,nLO ; turn on Low LED

;*** Test for high illumination
; set high input threshold
movlw 1<<VREN|0<<VRR|.11 ; configure voltage reference:
; enable voltage reference (VREN = 1)
; CVref = 0.594*Vdd (VRR = 0, VR = 11)
movwf VRCON ; -> CVref = 2.97 V
Delay10us ; wait 10 us to settle

; compare with input
btfsc CM2CON0,C2OUT ; if C2IN+ > CVref
bsf sPORTC,nHI ; turn on High LED

;*** Display test results
movf sPORTC,w ; copy shadow to PORTC
movwf PORTC

; repeat forever
goto main_loop

END

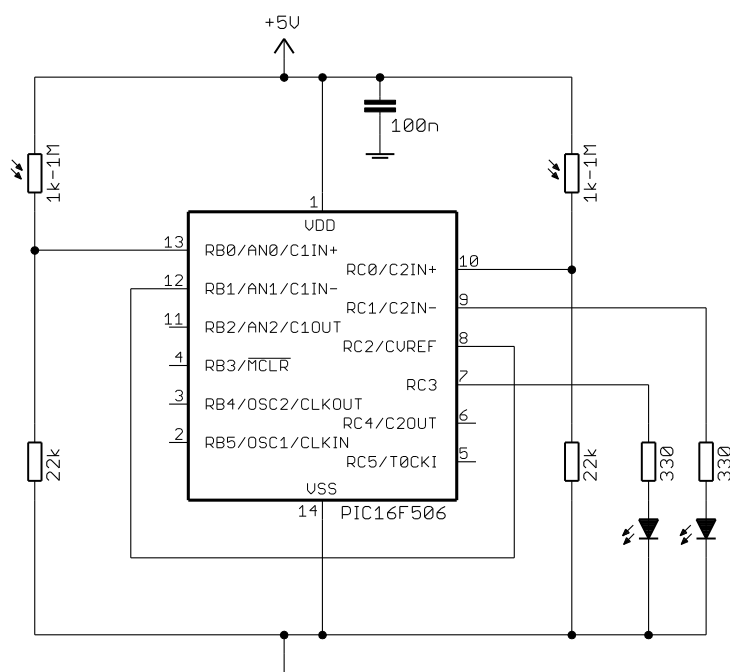
```

Using both comparators with the programmable voltage reference

For a final example, suppose that we want to test two input signals (say, light level in two locations) by comparing them against a common reference. We would need to use two comparators, with an input signal connected to each, and a single threshold voltage level connected to both.

What if we want to use the programmable voltage reference to generate the common threshold?

We've seen that CVREF cannot be selected as an input to comparator 1, so it would seem that it's not possible to use the programmable voltage reference with comparator 1.



But although no internal connection is available, that doesn't rule out an external connection – and as we saw above, the programmable reference can be made available on the CVREF pin.

So, to use the programmable voltage reference with comparator 1, we need to set the VROE bit in the VRCON register, to enable the CVREF output, and connect the CVREF pin to a comparator 1 input – as shown in the circuit diagram on the left, where CVREF is connected to C1IN-.

If you are using the [Gooligum baseline training board](#), you can keep the board set up as before, with shunts in JP17, JP19, and position 2 ('C2IN+') of JP25, and add a shunt across pins 2 and 3 ('LDR1') of JP24, to also connect photocell PH1 to C1IN+. You also need

to connect CVREF to C1IN-, which you can do by linking pins 9 ('GP/RA/RB1') and 11 ('RC2') on the 16-pin header.

If you are using Microchip's Low Pin Count Demo Board, the connection from CVREF to C1IN- can be made by linking pins 8 and 12 on the 14-pin header, and the photocells and associated resistors can be connected via the 14-pin header as before.

Note: Whichever board you are using, you should disconnect your PICkit 2 or PICkit 3 from the board when you run the program (applying external power instead), because the programmer loads RB1/AN1/C1IN-, pulling down the reference voltage delivered by the CVREF pin. The circuit will still operate with a PICkit 2 or PICkit 3 connected, but the reference voltage will be much lower than it should be.

Most of the initialisation and main loop code is very similar to that used in earlier examples, but when configuring the voltage reference, we must ensure that the VROE bit is set:

```

movlw    1<<VREN|1<<VROE|1<<VRR|.12
                                ; CVref = 0.500*Vdd (VRR = 1, VR = 12)
                                ; enable CVref output pin (VROE = 1)
                                ; enable voltage reference (VREN = 1)
movwf    VRCON                  ; -> CVref = 2.50 V,
                                ;   CVref output pin enabled

```

Complete program

Here is the full listing for the "two inputs with a common programmed voltage reference" program:

```

;*****
;
; Description:      Lesson 9, example 5
;
; Demonstrates use of comparators 1 and 2
; with the programmable voltage reference
;
; Turns on: LED 1 when C1IN+ > 2.5 V
; and LED 2 when C2IN+ > 2.5 V
;
;*****
; Pin assignments:
; C1IN+ = input 1 (LDR/resistor divider)
; C1IN- = connected to CVref
; C2IN+ = input 2 (LDR/resistor divider)
; CVref = connected to C1IN-
; RC1 = indicator LED 2
; RC3 = indicator LED 1
;
;*****

list          p=16F506
#include      <p16F506.inc>

radix        dec

;***** CONFIGURATION
; ext reset, no code protect, no watchdog, 4 Mhz int clock
__CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IOSCFS_OFF & _IntRC_OSC_RB4EN

```

```

; pin assignments
    constant    nLED1=RC3          ; indicator LED 1
    constant    nLED2=RC1          ; indicator LED 2

;***** VARIABLE DEFINITIONS
        UDATA_SHR
sPORTC   res 1                      ; shadow copy of PORTC

;***** RC CALIBRATION
RCCAL   CODE    0x3FF              ; processor reset vector
        res 1                      ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET   CODE    0x000              ; effective reset vector
        movwf   OSCCAL             ; apply internal RC factory calibration

;***** MAIN PROGRAM *****

;***** Initialisation
start
    ; configure ports
    movlw    ~(1<<nLED1|1<<nLED2)    ; configure PORTC LED pins as outputs
    tris    PORTC

    ; configure comparator 1
    movlw   1<<C1PREF|1<<C1NREF|1<<C1POL|1<<C1ON
                ; +ref is C1IN+ (C1PREF = 1)
                ; -ref is C1IN- (C1NREF = 1)
                ; normal polarity (C1POL = 1)
                ; comparator on (C1ON = 1)
    movwf   CM1CON0                ; -> C1OUT = 1 if C1IN+ > C1IN- (= CVref)

    ; configure comparator 2
    movlw   1<<C2PREF1|0<<C2NREF|1<<C2POL|1<<C2ON
                ; +ref is C2IN+ (C2PREF1 = 1)
                ; -ref is CVref (C2NREF = 0)
                ; normal polarity (C2POL = 1)
                ; comparator on (C2ON = 1)
    movwf   CM2CON0                ; -> C2OUT = 1 if C2IN+ > CVref

    ; configure voltage reference
    movlw   1<<VREN|1<<VROE|1<<VRR|.12
                ; CVref = 0.500*Vdd (VRR = 1, VR = 12)
                ; enable CVref output pin (VROE = 1)
                ; enable voltage reference (VREN = 1)
    movwf   VRCON                  ; -> CVref = 2.50 V,
                ; CVref output pin enabled

;***** Main loop
main_loop
    ; start with shadow PORTC clear
    clrf    sPORTC

    ; test input 1
    btfsc   CM1CON0,C1OUT          ; if C1IN+ > CVref
    bsf     sPORTC,nLED1          ; turn on LED 1

```

```
; test input 2
btfsc  CM2CON0,C2OUT      ; if C2IN+ > CVref
bsf    sPORTC,nLED2      ; turn on LED 2

; display test results
movf   sPORTC,w          ; copy shadow to PORTC
movwf  PORTC

; repeat forever
goto  main_loop

END
```

Conclusion

This has been a long lesson, for such an apparently simple peripheral.

As we've seen, the comparators on baseline PICs can be configured with flexible combinations of inputs, including an absolute voltage reference and a programmable voltage reference (which can be made available externally). We've also seen how to use the external comparator outputs to generate hysteresis, and how comparator 1 can be used to clock the timer – which, as we demonstrated, can be used to count pulses that would be otherwise unsuited to digital inputs.

The [next lesson](#) continues the topic of analog inputs on baseline PICs, with an overview of analog-to-digital conversion.

Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 10: Analog-to-Digital Conversion

We saw in the [last lesson](#) how a comparator can be used to respond to an analog signal being above or below a specific threshold. In other cases, the value of the input is important and you need to measure, or *digitise* it, so that your code can process a digital representation of the signal's value.

This lesson explains how to use the analog-to-digital converter (ADC), available on a number of baseline PICs, to read analog inputs, converting them to digital values you can operate on.

To display these values, we'll make use of the 7-segment displays used in [lesson 8](#).

In summary, this lesson covers:

- Using an ADC module to read analog inputs
- Hexadecimal output on 7-segment displays

Analog-to-Digital Converter

The analog-to-digital converter (ADC) on the 16F506 allows you to measure analog input voltages to a resolution of 8 bits. An input of 0 V (or VSS, if VSS is not at 0 V) will read as 0, while an input of VDD corresponds to the full-scale reading of 255.

Three analog input pins are available: AN0, AN1 and AN2. But, since there is only one ADC module, only one input can be read (or *converted*) at once.

The analog-to-digital converter is controlled by the ADCON0 register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADCON0	ANS1	ANS0	ADCS1	ADCS0	CHS1	CHS0	GO/ $\overline{\text{DONE}}$	ADON

Before a pin can be selected as an input channel for the ADC, it must first be configured as an analog input, using the ANS<1:0> bits:

ANS<1:0>	Pins configured as analog inputs
00	none
01	AN2 only
10	AN0 and AN2
11	AN0, AN1 and AN2

Note that pins cannot be independently configured as analog inputs.

If only one analog input is needed, it has to be AN2. If any analog inputs are configured, AN2 must be one of them.

If only two analog inputs are needed, they must be AN0 and AN2.

By default, following power-on, $ANS<1:0>$ is set to '11', configuring AN0, AN1 and AN2 as analog inputs.

This is the default behaviour for all PICs; all pins that can be configured as analog inputs will be configured as analog inputs at power-on, and you must explicitly disable the analog configuration on a pin if you wish to use it for digital I/O. This is because, if a pin is configured as a digital input, it will draw excessive current if the input voltage is not at a digital "high" or "low" level, i.e. somewhere in-between. Thus, the safe, low-current option is to default to analog inputs and to leave it up to your program to only enable digital inputs on those pins known to be digital.

All of the analog inputs can be disabled (enabling digital I/O on the RB0, RB1 and RB2 pins) by clearing $ADCON0$, which clears $ANS<1:0>$ to '00'.

The $ADON$ bit turns the ADC module on or off: '1' to turn it on, '0' to turn it off. The ADC module is turned on ($ADON = 1$) by default, at power-on.

Note: To minimise power consumption, the ADC module should be turned off before entering sleep mode.

Hence, clearing $ADCON0$ will also clear $ADON$ to '0', disabling the ADC module, conserving power.

However, disabling the ADC module is not enough to disable the analog inputs; the $ANS<1:0>$ bits must be used to configure analog pins for digital I/O, regardless of the value of $ADON$.

The analog-to-digital conversion process is driven by a clock, which is derived from either the processor clock ($FOSC$) or the internal RC oscillator ($INTOSC$). For accurate conversions, the ADC clock rate must be selected such that the ADC conversion clock period, TAD , be between 500 ns and 50 μ s.

The ADC conversion clock is selected by the $ADCS<1:0>$ bits:

$ADCS<1:0>$	ADC conversion clock
00	$FOSC/16$
01	$FOSC/8$
10	$FOSC/4$
11	$INTOSC/4$

Note that, if the internal RC oscillator is being used as the processor clock, the $INTOSC/4$ and $FOSC/4$ options are the same.

But whether you are using a high-speed 20 MHz crystal, a low-power 32 kHz watch crystal, or a low-speed external RC oscillator, the $INTOSC/4$ ADC clock option ($ADCS<1:0> = '11'$) will always work, giving accurate conversions.

$INTOSC/4$ is always a safe choice.

Each analog-to-digital conversion requires 13 TAD periods to complete.

If you are using the $INTOSC/4$ ADC clock option, and the internal RC oscillator is running at 4 MHz, $INTOSC/4 = 1$ MHz and $TAD = 1$ μ s. Each conversion will then take a total of 13 μ s.

If the internal RC oscillator is running at 8 MHz, $TAD = 500$ ns (the shortest period allowed, making this the fastest conversion rate possible), each conversion will take 13×500 ns = 6.5 μ s.

Having turned on the ADC, selected the ADC conversion clock, and configured the analog input pins, the next step is to select an input (or *channel*) to be converted.

CHS<1:0> selects the ADC channel:

CHS<1:0>	ADC channel
00	analog input AN0
01	analog input AN1
10	analog input AN2
11	0.6V internal voltage reference

Note that, in addition to the three analog input pins, AN0 to AN2, the 0.6V internal voltage reference can be selected as an ADC input channel.

Why measure the 0.6V absolute reference voltage, if it never changes?

That’s the point – it never changes (except for some drift with temperature). But if you’re not using a regulated power supply (e.g. running direct from batteries), VDD will vary as the power supply changes. Since the full scale range of the ADC is VSS to VDD, your analog measurements are a fraction of VDD and will vary as VDD varies. By regularly measuring (*sampling*) the 0.6 V absolute reference, it is possible to achieve greater measurement accuracy by correcting for changes in VDD. It is also possible to use the 0.6 V reference to indirectly measure VDD, and hence battery voltage, as we’ll see in a later example.

Having set up the ADC and selected an input channel to be sampled, the final step is to begin the conversion, by setting the GO/ DONE bit to ‘1’.

Your code then needs to wait until the GO/ DONE bit has been cleared to ‘0’, which indicates that the conversion is complete. You can then read the conversion result from the ADRES register.

You should copy the result from ADRES before beginning the next conversion, so that it isn’t overwritten during the conversion process¹.

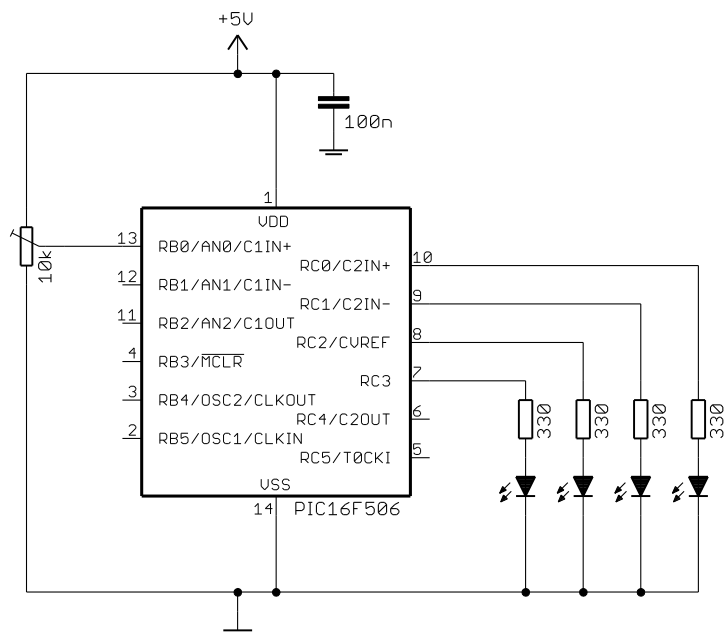
Also for best results, the source impedance of the input being sampled should be no more than 10 kΩ.

Example 1: Binary Output

As a simple demonstration of how to use the ADC, we can use a potentiometer to provide a variable voltage to an analog input, and four LEDs to show a 4-bit binary representation of that value, using the circuit shown on the right.

To implement it using the [Gooligum baseline training board](#), place a shunt across pins 1 and 2 (‘POT’) of JP24, connecting the 10 kΩ pot (RP2) to AN0, and shunts in JP16-19, enabling the LEDs on RC0-3.

If you are using Microchip’s Low Pin Count Demo Board, the onboard pot and LEDs are already connected to AN0 and RC0 – RC3. You only need to ensure that jumpers JP1-5 are closed.



¹ The result actually remains in ADRES for the first four TAD periods after the conversion begins. This is the sampling period, and for best results the input signal should not be changing rapidly during this period.

To make the display meaningful (i.e. a binary representation of the input voltage, corresponding to sixteen input levels), the top four bits of the ADC result (in ADRES) should be copied to the four LEDs.

The bottom four bits of the ADC result are thrown away; they are not significant.

To use RC0 and RC1 as digital outputs, we need to disable the C2IN+ and C2IN- inputs, which can be done by disabling comparator 2:

```
clrf    CM2CON0           ; disable comparator 2 -> RC0, RC1 digital
```

This also disables C2OUT, making RC4 available for digital I/O, even though it isn't used in this example.

To use RC2 as a digital output, the CVREF output has to be disabled. By default, on power-up, the voltage reference module is disabled, including the CVREF output. But it doesn't hurt to explicitly disable it as part of your initialisation code:

```
clrf    VRCON             ; disable CVref -> RC2 usable
```

AN0 has to be configured as an analog input. It's not possible to configure AN0 as an analog input without AN2, so for the minimal number of analog inputs, set ANS<1:0> = '10' (AN0 and AN2 analog).

It makes sense to choose INTOSC/4 as the conversion clock (ADCS<1:0> = '11'), as a safe default, although in fact any of the ADC clock settings will work when the processor clock is 4 MHz or 8 MHz.

AN0 has to be selected as the ADC input channel: CHS<1:0> = '00'.

And of course the ADC module has to be enabled: ADON = '1'.

So we have:

```
movlw   b'10110001'      ; configure ADC:
        ; 10-----      AN0, AN2 analog (ANS = 10)
        ; --11-----     clock = INTOSC/4 (ADCS = 11)
        ; ----00--       select channel AN0 (CHS = 00)
        ; -----1       turn ADC on (ADON = 1)
movwf   ADCON0           ; -> AN0 ready for sampling
```

Alternatively the 'movlw' could be written as:

```
movlw   b'10'<<ANS0|b'11'<<ADCS0|b'00'<<CHS0|1<<ADON
```

But that's unwieldy, and harder to understand.

Having configured the LED outputs (PORTC) and the ADC, the main loop is quite straightforward:

```
main_loop
    ; sample analog input
    bsf    ADCON0,GO      ; start conversion
w_adc   btfsf  ADCON0,NOT_DONE ; wait until done
        goto   w_adc

        ; display result on 4 x LEDs
    swapf  ADRES,w        ; copy high nybble of result
    movwf  LEDS           ; to low nybble of output port (LEDs)

        ; repeat forever
    goto   main_loop
```

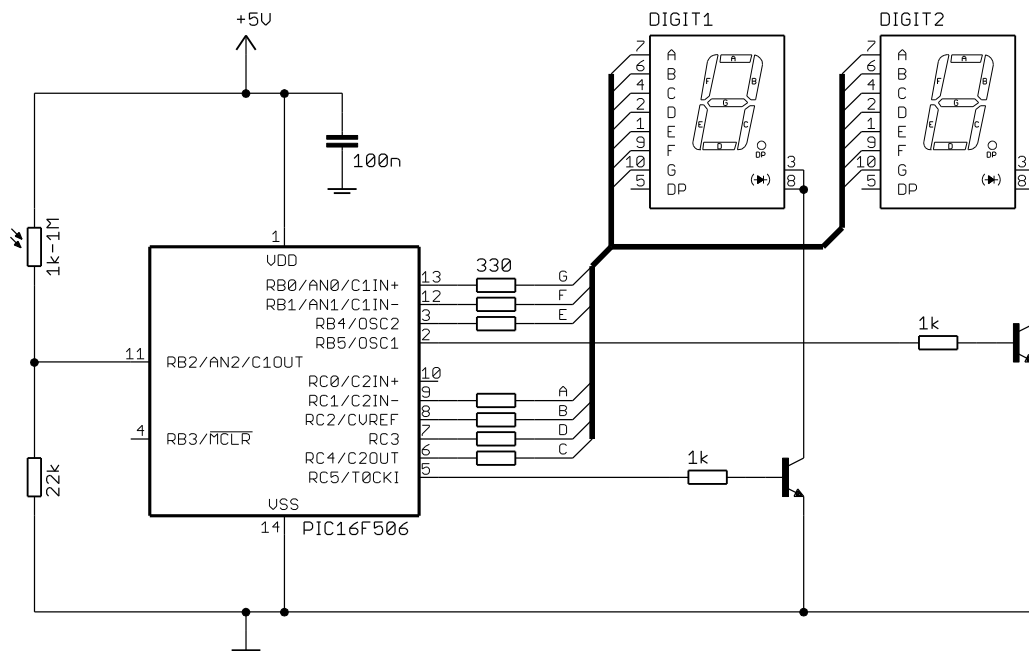
You'll see that two symbols are used for the $\overline{GO/DONE}$ bit, depending on the context: when setting the bit to start the conversion, it is referred to as "GO", but when using it as a flag to check whether the conversion is complete, it is referred to as "NOT_DONE".

Using the appropriate symbol for the context makes the intent of the code clearer, even though both symbols refer the same bit.

Finally, note the use of the 'swapf' instruction. The output bits we need to copy are in the high nybble of ADRES, while the output LEDs (RC0 – RC3) form the low nybble of PORTC, making 'swapf' a neat solution; much shorter than using four right-shifts.

Example 2: Hexadecimal Output

A binary LED display, as in example 1, is not a very useful form of output. To create a more human-readable output, we can modify the multi-digit 7-segment LED circuit from [lesson 8](#) by dropping one digit, and adding a photocell and resistor to supply a voltage that increases with light level (as we saw in [lesson 9](#)), as shown below:



To implement this circuit using the [Gooligum baseline training board](#), place shunts:

- across every position (all six of them) of jumper block JP4, connecting segments A-D, F and G to pins RB0-1 and RC1-4
- in position 1 ('RA/RB4') of JP5, connecting segment E to pin RB4
- across pins 2 and 3 ('RC5') of JP6, connecting digit 1 to the transistor controlled by RC5
- in jumpers JP8 and JP9, connecting pins RC5 and RB5 to their respective transistors
- in position 1 ('AN2') of JP25, connecting photocell PH2 to AN2.

All other shunts should be removed.

If you are using Microchip's Low Pin Count Demo Board, you will need to supply your own display modules, resistors, transistors and photocell, and connect them to the PIC via the 14-pin header on that board, as described in [lessons 8](#) and [9](#).

To display a hexadecimal value representing the light level, we can adapt the multiplexed 7-segment display code from [lesson 8](#).

First, to drive the displays using RC1-RC5 and RB0, RB1, RB4 and RB5, we need to disable the comparators, comparator outputs, and voltage reference output:

```

; configure ports
clrw                ; configure PORTB and PORTC as all outputs
tris    PORTB
tris    PORTC
clrf   CM1CON0      ; disable comparator 1 -> RB0, RB1 digital
clrf   CM2CON0      ; disable comparator 2 -> RC0, RC1 digital
clrf   VRCON        ; disable CVref -> RC2 usable

```

To use RB0 and RB1 for digital I/O, it is also necessary to deselect AN0 and AN1 as analog inputs, configuring only AN2 as an analog input with ANS = 01.

Since we are using AN2 as an analog input, we need to select it as the active ADC input with CHS = 10.

So, to configure and select only AN2 as an analog input, we initialise the ADC using:

```

movlw    b'01111001'    ; configure ADC:
            ; 01-----    AN2 (only) analog (ANS = 01)
            ; --11-----    clock = INTOSC/4 (ADCS = 11)
            ; ----10--    select channel AN2 (CHS = 10)
            ; -----1    turn ADC on (ADON = 1)
movwf    ADCON0          ; -> AN2 ready for sampling

```

As we did in [lesson 8](#), the timer is used to provide a ~2 ms tick to drive the display multiplexing:

```

; configure timer
movlw    b'11010111'    ; configure Timer0:
            ; --0-----    timer mode (T0CS = 0) -> RC5 usable
            ; ----0---    prescaler assigned to Timer0 (PSA = 0)
            ; -----111    prescale = 256 (PS = 111)
option   ; -> increment every 256 us
            ; (TMR0<2> cycles every 2.048 ms)

```

This assumes a 4 MHz clock, not 8 MHz, so the configuration directive needs to include ‘_IOSCF5_OFF’:

```

; ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IOSCF5_OFF & _IntRC_OSC_RB4EN

```

Note also that, by clearing the T0CS bit, placing the timer in timer mode, the T0CKI input (see [lesson 5](#)) is disabled, making RC5 (which, on the PIC16F506, shares its pin with T0CKI) available as a digital output. So, even if we weren’t using Timer0 in this example, we’d still have to clear T0CS, to make it possible to use RC5 as an output.

The lookup tables have to be extended to include 7-segment representations of the letters ‘A’ to ‘F’, but the lookup code remains the same as in [lesson 8](#).

Since each digit is displayed for 2 ms, and the analog to digital conversion only takes around 13 µs, the ADC read can be completed well within the time spent waiting to begin displaying the next digit (~1 ms), without affecting the display multiplexing.

The main loop, then, simply consists of reading the analog input and displaying each digit of the result, then repeating that quickly enough for the display to appear to be continuous:

```
main_loop
    ; sample input
    bsf    ADCON0,GO        ; start conversion
w_adc    btfsc   ADCON0,NOT_DONE ; wait until conversion complete
        goto    w_adc

        ; display high nybble for 2.048 ms
w10_hi   btfss   TMR0,2        ; wait for TMR0<2> to go high
        goto    w10_hi
        swapf   ADRES,w        ; get "tens" digit
        andlw   0x0F          ; from high nybble of ADC result
        [code to display "tens" digit then wait for TMR<2> to go low goes here]

        ; display ones for 2.048 ms
w1_hi    btfss   TMR0,2        ; wait for TMR0<2> to go high
        goto    w1_hi
        movf    ADRES,w        ; get ones digit
        andlw   0x0F          ; from low nybble of ADC result
        [code to display "ones" digit then wait for TMR<2> to go low goes here]

        ; repeat forever
        goto    main_loop
```

Complete program

Here is the complete “hexadecimal light meter”, so that you can see where and how the various program fragments fit in:

```
*****
;
; Description:    Lesson 10, example 2
;
; Displays ADC output in hexadecimal on 7-segment LED displays
;
; Continuously samples analog input,
; displaying result as 2 x hex digits on multiplexed 7-seg displays
;
;*****
;
; Pin assignments:
; AN2           = voltage to be measured (e.g. pot or LDR)
; RB0-1,RB4,RC1-4 = 7-segment display bus (common cathode)
; RC5           = "tens" digit enable (active high)
; RB5           = ones digit enable
;
;*****

list          p=16F506
#include       <p16F506.inc>

radix        dec

;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG      _MCLRE_ON & _CP_OFF & _WDT_OFF & _IOSCFS_OFF & _IntRC_OSC_RB4EN
```

```

; pin assignments
  #define TENS    PORTC,5    ; "tens" digit enable
  #define ONES   PORTB,5    ; ones digit enable

;***** VARIABLE DEFINITIONS
      UDATA_SHR
temp   res 1                ; used by set7seg routine (temp digit store)

;***** RC CALIBRATION
RCCAL  CODE    0x3FF        ; processor reset vector
      res 1                ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET  CODE    0x000        ; effective reset vector
      movwf   OSCCAL        ; apply internal RC factory calibration
      pagesel start
      goto    start        ; jump to main code

;***** Subroutine vectors
set7seg
      pagesel set7seg_R
      goto    set7seg_R

;***** MAIN PROGRAM *****
MAIN   CODE

;***** Initialisation
start
      ; configure ports
      clrw                    ; configure PORTB and PORTC as all outputs
      tris   PORTB
      tris   PORTC
      clrf   CM1CON0        ; disable comparator 1 -> RB0, RB1 digital
      clrf   CM2CON0        ; disable comparator 2 -> RC0, RC1 digital
      clrf   VRCON          ; disable CVref -> RC2 usable

      ; configure ADC
      movlw  b'01111001'    ; configure ADC:
      ; 01-----          AN2 (only) analog (ANS = 01)
      ; --11-----         clock = INTOSC/4 (ADCS = 11)
      ; ----10--          select channel AN2 (CHS = 10)
      ; -----1          turn ADC on (ADON = 1)
      movwf  ADCON0        ; -> AN2 ready for sampling

      ; configure timer
      movlw  b'11010111'    ; configure Timer0:
      ; --0-----         timer mode (T0CS = 0) -> RC5 usable
      ; ----0---          prescaler assigned to Timer0 (PSA = 0)
      ; -----111        prescale = 256 (PS = 111)
      option                    ; -> increment every 256 us
      ; (TMR0<2> cycles every 2.04 8ms)

;***** Main loop
main_loop
      ; sample input
      bsf   ADCON0,GO        ; start conversion
w_adc  btfs  ADCON0,NOT_DONE ; wait until conversion complete
      goto  w_adc

```

```

; display high nybble for 2.048 ms
w10_hi  btfss   TMR0,2           ; wait for TMR0<2> to go high
        goto    w10_hi
        swapf   ADRES,w         ; get "tens" digit
        andlw   0x0F           ; from high nybble of ADC result
        pagesel set7seg
        call    set7seg         ; then output it
        pagesel $
        bsf     TENS            ; enable "tens" display
w10_lo  btfsc   TMR0,2           ; wait for TMR<2> to go low
        goto    w10_lo

; display ones for 2.048 ms
w1_hi   btfss   TMR0,2           ; wait for TMR0<2> to go high
        goto    w1_hi
        movf    ADRES,w         ; get ones digit
        andlw   0x0F           ; from low nybble of ADC result
        pagesel set7seg
        call    set7seg         ; then output it
        pagesel $
        bsf     ONES           ; enable ones display
w1_lo   btfsc   TMR0,2           ; wait for TMR<2> to go low
        goto    w1_lo

; repeat forever
        goto    main_loop

;***** LOOKUP TABLES *****
TABLES  CODE    0x200           ; locate at beginning of a page

; pattern table for 7 segment display on port B
; RB4 = E, RB1:0 = FG
get7sB  addwf   PCL,f
        retlw  b'010010'       ; 0
        retlw  b'000000'       ; 1
        retlw  b'010001'       ; 2
        retlw  b'000001'       ; 3
        retlw  b'000011'       ; 4
        retlw  b'000011'       ; 5
        retlw  b'010011'       ; 6
        retlw  b'000000'       ; 7
        retlw  b'010011'       ; 8
        retlw  b'000011'       ; 9
        retlw  b'010011'       ; A
        retlw  b'010011'       ; b
        retlw  b'010010'       ; C
        retlw  b'010001'       ; d
        retlw  b'010011'       ; E
        retlw  b'010011'       ; F

; pattern table for 7 segment display on port C
; RC4:1 = CDBA
get7sC  addwf   PCL,f
        retlw  b'011110'       ; 0
        retlw  b'010100'       ; 1
        retlw  b'001110'       ; 2
        retlw  b'011110'       ; 3
        retlw  b'010100'       ; 4
        retlw  b'011010'       ; 5
        retlw  b'011010'       ; 6

```

```

    retlw    b'010110'    ; 7
    retlw    b'011110'    ; 8
    retlw    b'011110'    ; 9
    retlw    b'010110'    ; A
    retlw    b'011000'    ; b
    retlw    b'001010'    ; C
    retlw    b'011100'    ; d
    retlw    b'001010'    ; E
    retlw    b'000010'    ; F

; Display digit passed in W on 7-segment display
set7seg_R
    ; disable displays
    clrf    PORTB        ; clear all digit enable lines on PORTB
    clrf    PORTC        ; and PORTC

    ; output digit pattern
    movwf   temp        ; save digit
    call    get7sB      ; lookup pattern for port B
    movwf   PORTB       ; then output it
    movf    temp,w      ; get digit
    call    get7sC      ; then repeat for port C
    movwf   PORTC
    retlw   0

END

```

Of course, most people are more comfortable with a decimal output, perhaps 0-99, instead of hexadecimal.

And you'll find, if you build this as a light meter, using an LDR (CdS photocell), that although the output is quite stable when lit by daylight, the least significant digit jitters badly when the LDR is lit by incandescent and, in particular, fluorescent lighting. This is because these lights flicker at 50 or 60 Hz (depending on where you live); too quickly for your eyes to detect, but not too fast for this light meter to react to, since it is sampling and updating the display 244 times per second.

So some obvious improvements to the design would be to scale and display the output as 0-99 in decimal, and to smooth or filter high-frequency noise, such as that caused by fluorescent lighting.

We'll make those improvements in [lesson 11](#). But first we'll look at one last example.

Example 3: Measuring Supply Voltage

As mentioned above, the 0.6 V absolute voltage reference can be sampled by the ADC, and this provides a way to infer the supply voltage (actually $V_{DD} - V_{SS}$, but to keep this simple we'll assume $V_{SS} = 0$ V).

Assuming that $V_{DD} = 5.0$ V and $V_{SS} = 0$ V, the 0.6 V reference should read as:

$$0.6 \text{ V} \div 5.0 \text{ V} \times 255 = 30$$

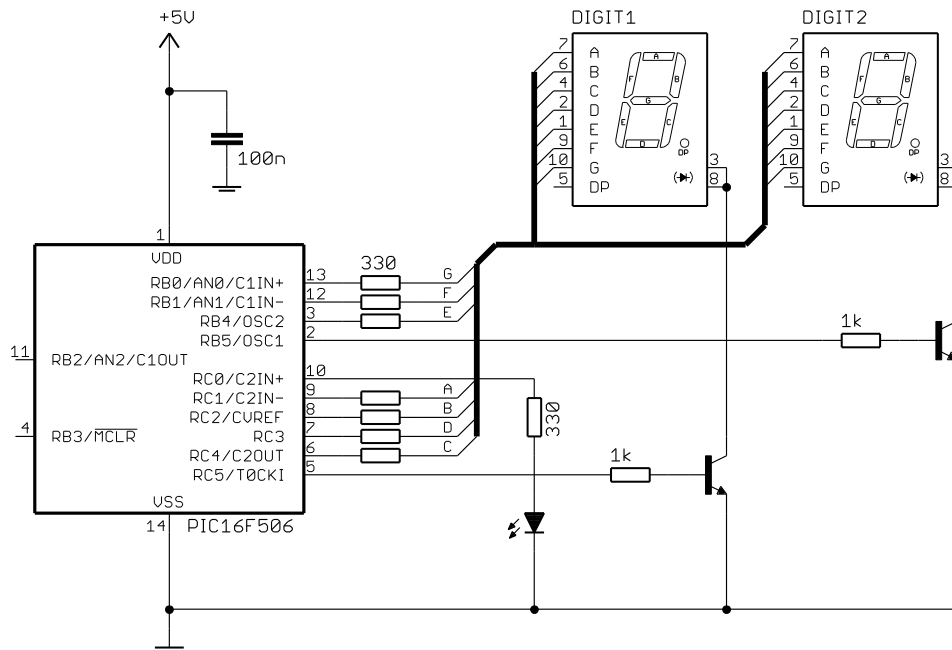
Now if V_{DD} was to fall to, say, 3.5 V, the 0.6 V reference will read as:

$$0.6 \text{ V} \div 3.5 \text{ V} \times 255 = 43$$

As V_{DD} falls, the 0.6 V reference will give a larger ADC result, since it remains constant as V_{DD} decreases.

So to check for the power supply falling too low, the value returned by sampling the 0.6 V reference can be compared with a threshold. For example, a value above 43 indicates that $V_{DD} < 3.5$ V, and perhaps a warning should be displayed, or the device shut down before power falls too low.

To illustrate this, we can use adapt the circuit and program from example 2, displaying the ADC reading corresponding to the 0.6 V reference as two hex digits, and light a “low voltage warning” LED attached to RC0 (as shown below).



If you are using the [Gooligum baseline training board](#), you should set it up as in the last example, but remove the shunt from JP25 (disconnecting the photocell from AN2) and close JP16 (connecting the LED on RC0).

To implement this low voltage warning, the code from example 2 can be used with very little modification.

To make the code easier to maintain, we can define the voltage threshold as a constant:

```
constant MINVDD=3500           ; Minimum Vdd (in mV)
constant VRMAX=255*600/MINVDD ; Threshold for 0.6 V ref measurement
```

Note that, because MPASM only supports integer expressions, “MINVDD” has to be expressed in millivolts instead of volts (so that fractions of a volt can be specified).

The initialisation code remains the same, except that the ADC configuration is changed to disable all the analog inputs and selecting the internal 0.6 V reference as the ADC input channel:

```
movlw   b'00111101'           ; configure ADC:
        ; 00-----           no analog inputs (ANS = 00) -> RB0-2 digital
        ; --11-----          clock = INTOSC/4 (ADCS = 11)
        ; ----11--            select 0.6 V reference (CHS = 11)
        ; -----1            turn ADC on (ADON = 1)
movwf   ADCON0                 ; -> 0.6 V reference ready for sampling
```

After sampling the 0.6 V input, we can test for VDD being too low by comparing the conversion result (ADRES) with the threshold (VRMAX):

```
        ; sample 0.6 V reference
        bsf   ADCON0,GO        ; start conversion
w_adc   btfs   ADCON0,NOT_DONE ; wait until conversion complete
        goto  w_adc
```

```

; test for low Vdd (measured 0.6 V > threshold)
movlw   VRMAX
subwf   ADRES,w           ; if ADRES > VRMAX
btfsc   STATUS,C
bsf     WARN              ; turn on warning LED

; display high nybble for 2.048 ms
[wait for TMR0<2> high then display "tens" digit]

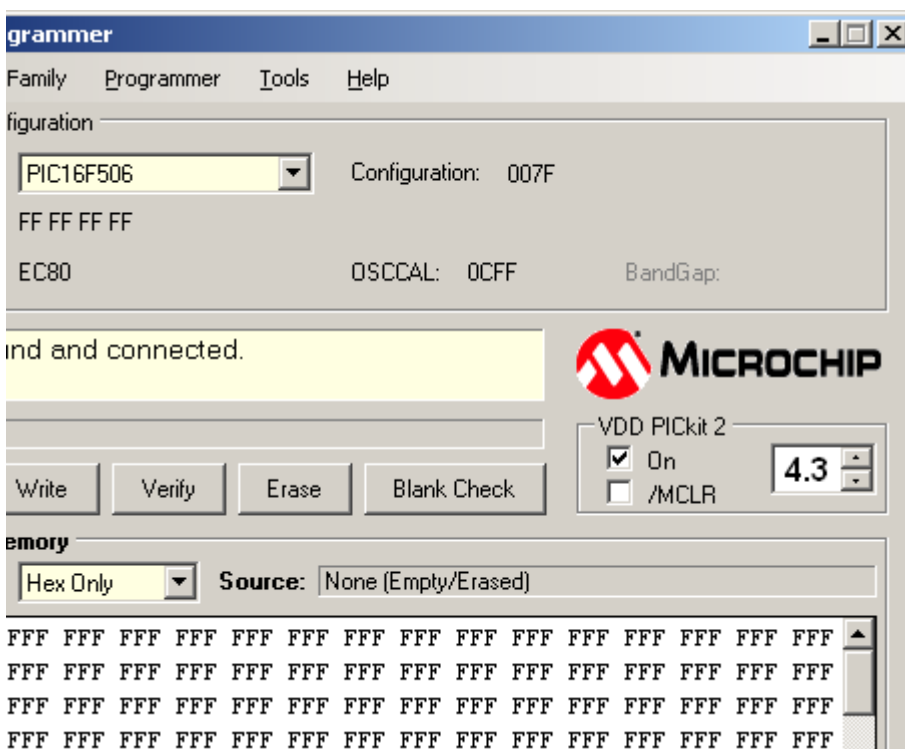
```

There's a slight problem with this approach: when a digit is displayed, the LED on RC0 will be extinguished, since the lookup table for PORTC always returns a '0' for bit 0. To avoid that problem, the 'set7seg_R' routine would need to be modified to include logical masking operations so that RC0 is not overwritten. But it's not really a significant problem; the LED will remain lit for ~1 ms, while the "display high nybble" routine waits for TMR0<2> to go high, out of a total multiplex cycle time of ~4 ms. That is, when the LED is 'lit', it will actually be on for ~25% of the time, and that's enough to make it visible.

To test this application, you need to be able to vary VDD.

If you are using a PICKit 2 or PICKit 3 to power your circuit, you can use the standalone PICKit 2 or PICKit 3 programming application (each downloadable from www.microchip.com) to vary VDD, while the circuit is powered. But first, you should exit MPLAB, so that you don't have two applications trying to control the PICKit 2 at once.

Although the PICKit 2 Programmer application is shown below, the PICKit 3 version looks almost identical, and the method for varying target power (VDD) is the same for both.



In the programmer application, select the Baseline device family, then the PIC16F506 device and then click 'On', as illustrated.

Your circuit should now be powered on, and, assuming the supply voltage is 5.0 V, the display should show '1E' (hexadecimal for 30), or something close to that.

You can now start to decrease VDD, by clicking on the down arrow next to the voltage display, 0.1 V at a time.

When you get to 3.5 V, the display should read '2b' (hex for 43) – but note that the PICKit 2 does not deliver as accurate a voltage as the PICKit 3, so if you are using a PICKit 2, you may see different values at "3.5 V".

When the voltage is low enough for the display to read '2b', the warning LED should light.

Conclusion

We've seen that it's relatively simple to setup and use the ADC on the PIC16F506, whether for the usual purpose of reading an analog quantity, or even to infer the PIC's supply voltage.

However, as mentioned earlier, the light meter project would be more useful if the output was converted to a range of 0-99 and displayed in decimal, and if the results were filtered to smooth out short term fluctuations.

The [next lesson](#) will complete our overview of baseline PIC assembler, by demonstrating how to perform some simple arithmetic operations, including moving averages and working with arrays, to implement these suggested improvements.

Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 11: Integer Arithmetic and Arrays

In the [last lesson](#), we saw how to read an analog input and display the “raw” result. But in most cases the raw values aren’t directly usable; normally they will need to be processed in some way before being displayed or used in decision making. While advanced signal processing is beyond the capabilities of baseline and even midrange PICs, this lesson demonstrates that basic post-processing, such as integer scaling and simple filtering, can be readily accomplished with even the lowest-end PICs.

This lesson introduces some of the basic integer arithmetic operations. For more complete coverage of this topic, refer to Microchip’s application notes *AN526: “PIC16C5X / PIC16CXXX Math Utility Routines”*, and *AN617: “Fixed Point Routines”*, available at www.microchip.com.

We’ll also see how to use indirect addressing to implement arrays, illustrated by a simple moving average routine, used to filter noise from an analog signal.

In summary, this lesson covers:

- Multi-byte (including 16-bit and 32-bit) addition and subtraction
- Two’s complement representation of negative numbers
- 8-bit unsigned multiplication
- Using indirect addressing to work with arrays
- Calculating a moving average

Integer Arithmetic

At first sight, the baseline PICs seem to have very limited arithmetic capabilities: just a single 8-bit addition instruction (`addwf`) and a single 8-bit subtraction instruction (`subwf`).

However, addition and subtraction can be extended to arbitrarily large numbers by using the carry flag (`C`, in the `STATUS` register), which indicates when a result cannot be represented in a single 8-bit byte.

The `addwf` instruction sets the carry flag if the result *overflows* a single byte, i.e. is greater than 255.

And as explained in [lesson 5](#), the carry flag acts as a “not borrow” in a subtraction: the `subwf` instruction clears `C` if a borrow occurs, i.e. the result is negative.

The carry flag allows us to cascade addition or subtraction operations when working with long numbers.

Multi-byte variables

To store values larger than 8-bits, you need to allocate multiple bytes of memory to each, for example:

```

        UDATA
a       res 2                ; 16-bit variables "a" and "b"
b       res 2

```

You must then decide how to order the bytes within the variable – whether to place the least significant byte at the lowest address in the variable (known as *little-endian* ordering) or the highest (*big-endian*).

For example, to store the number 0x482C in variable “a”, the bytes 0x48 and 0x2C would be placed in memory as shown:

	a	a+1
Little-endian	0x2C	0x48
Big-endian	0x48	0x2C

Big-endian ordering has the advantage of making values easy to read in a hex dump, where increasing addresses are presented left to right. On the other hand, little-endian ordering makes a certain sense, because increasing addresses store increasingly significant bytes.

Which ordering you chose is entirely up to you; both are valid. This tutorial uses little-endian ordering, but the important thing is to be consistent.

16-bit addition

The following code adds the contents of the two 16-bit variables, “a” and “b”, so that $b = b + a$, assuming little-endian byte ordering:

```

movf    a,w           ; add LSB
addwf   b,f
btfsc   STATUS,C      ; increment MSB if carry
incf    b+1,f
movf    a+1,w         ; add MSB
addwf   b+1,f

```

After adding the least significant bytes (LSB’s), the carry flag is checked, and, if the LSB addition overflowed, the most significant byte (MSB) of the result is incremented, before the MSB’s are added.

Multi-byte (including 32-bit) addition

It may appear that this approach would be easily extended to longer numbers by testing the carry after the final ‘addwf’, and incrementing the next MSB of the result if carry was set. But there’s a problem. What if the LSB addition overflows, while (b+1) contains \$FF? The ‘incf b+1, f’ instruction will increment (b+1) to \$00, which should result in a “carry”, but it doesn’t, since ‘incf’ does not affect the carry flag.

By re-ordering the instructions, it is possible to use the ‘incfsz’ instruction to neatly avoid this problem:

```

movf    a,w           ; add LSB
addwf   b,f
movf    a+1,w         ; get MSB(a)
btfsc   STATUS,C      ; if LSB addition overflowed,
incfsz  a+1,w         ; increment copy of MSB(a)
addwf   b+1,f         ; add to MSB(b), unless MSB(a) is zero

```

On completion, the carry flag will now be set correctly, allowing longer numbers to be added by repeating the final four instructions. For example, for a 32-bit add:

```

movf    a,w           ; add byte 0 (LSB)
addwf   b,f
movf    a+1,w         ; add byte 1
btfsc   STATUS,C
incfsz  a+1,w
addwf   b+1,f
movf    a+2,w         ; add byte 2
btfsc   STATUS,C
incfsz  a+2,w
addwf   b+2,f
movf    a+3,w         ; add byte 3 (MSB)
btfsc   STATUS,C
incfsz  a+3,w
addwf   b+3,f

```

Multi-byte (including 16-bit and 32-bit) subtraction

Long integer subtraction can be done using a very similar approach.

For example, to subtract the contents of the two 16-bit variables, “a” and “b”, so that $b = b - a$, assuming little-endian byte ordering:

```

movf    a,w          ; subtract LSB
subwf   b,f
movf    a+1,w        ; get MSB(a)
btfss   STATUS,C     ; if borrow from LSB subtraction,
incfsz  a+1,w        ; increment copy of MSB(a)
subwf   b+1,f        ; subtract MSB(b), unless MSB(a) is zero

```

This approach is readily extended to longer numbers, by repeating the final four instructions.

For a 32-bit subtraction, we have:

```

movf    a,w          ; subtract byte 0 (LSB)
subwf   b,f
movf    a+1,w        ; subtract byte 1
btfss   STATUS,C
incfsz  a+1,w
subwf   b+1,f
movf    a+2,w        ; subtract byte 2
btfss   STATUS,C
incfsz  a+2,w
subwf   b+2,f
movf    a+3,w        ; subtract byte 3 (MSB)
btfss   STATUS,C
incfsz  a+3,w
subwf   b+3,f

```

Two's complement

Microchip's application note AN526 takes a different approach to subtraction.

Instead of subtracting a number, it is *negated* (made negative), and then added. That is, $b - a = b + (-a)$.

Negating a binary number is also referred to as taking its *two's complement*, since the operation is equivalent to subtracting it from a power of two.

The two's complement of an n-bit number, “a”, is given by the formula $2^n - a$.

For example, the 8-bit two's complement of 10 is $2^8 - 10 = 256 - 10 = 246$.

The two's complement of a number acts the same as a negative number would, in fixed-length binary addition and subtraction.

For example, $10 + (-10) = 0$ is equivalent to $10 + 246 = 256$, since in an 8-bit addition, the result (256) overflows, giving an 8-bit result of 0.

Similarly, $10 + (-9) = 1$ is equivalent to $10 + 247 = 257$, which overflows, giving an 8-bit result of 1.

And $10 + (-11) = -1$ is equivalent to $10 + 245 = 255$, which is the two's complement of 1.

Thus, two's complement is normally used to represent negative numbers in binary integer arithmetic, because addition and subtraction continue to work the same way. The only thing that needs to change is how the numbers being added or subtracted, and the results, are interpreted.

For unsigned quantities, the range of values for an n-bit number is from 0 to $2^n - 1$.

For signed quantities, the range is from -2^{n-1} to $2^{n-1} - 1$.

For example, 8-bit signed numbers range from -128 to 127.

The usual method used to calculate the two's complement of a number is to take the ones' complement (flip all the bits) and then add one.

This method is used in the 16-bit negate routine provided in AN526:

```
neg_A   comf    a, f           ; negate a ( -a -> a )
        incf    a, f
        btfsc   STATUS, Z
        decf    a+1, f
        comf    a+1, f
```

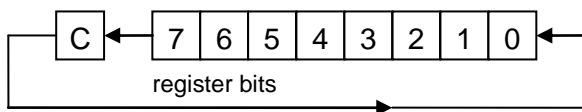
There is a new instruction here: 'comf f, d' – “**complement register file**”, which calculates the ones' complement of register 'f', placing the result back into the register if the destination is 'f', or in W if the destination is 'w'.

One reason you may wish to negate a number is to display it, if it is negative.

To test whether a two's complement signed number is negative, check its most significant bit, which acts as a sign bit: '1' indicates a negative number, '0' indicates non-negative (positive or zero).

Unsigned multiplication

It may seem that baseline PICs have no multiplication or division instructions, but that's not quite true: the “rotate left” instruction (rlf) can be used to shift the contents of a register one bit to the left, which has the effect of multiplying it by two:



Since the rlf instruction rotates bit 7 into the carry bit, and carry into bit 0, these instructions can be cascaded, allowing arbitrarily long numbers to be shifted left, and hence multiplied by two.

For example, to multiply the contents of 16-bit variable “a” by two, assuming little-endian byte ordering:

```
; left-shift 'a' (multiply by 2)
bcf    STATUS, C           ; clear carry
rlf    a, f               ; left shift LSB
rlf    a+1, f             ; then MSB (LSB<7> -> MSB<0> via carry)
```

[Although we won't consider division here (see AN526 for details), a similar sequence of “rotate right” instructions (rrf) can be used to shift an arbitrarily long number to the right, dividing it by two.]

You can see, then, that it is quite straightforward to multiply an arbitrarily long number by two. Indeed, by repeating the shift operation, multiplying or dividing by any power of two is easy to implement.

But that doesn't help us if we want to multiply by anything other than a power of two – or does it? Remember that every integer is composed of powers of two; that is how binary notation works

For example, the binary representation of 100 is 01100100 – the '1's in the binary number corresponding to powers of two:

$$100 = 64 + 32 + 4 = 2^6 + 2^5 + 2^2.$$

$$\text{Thus, } 100 \times N = (2^6 + 2^5 + 2^2) \times N = 2^6 \times N + 2^5 \times N + 2^2 \times N$$

In this way, multiplication by any integer can be broken down into a series of multiplications by powers of two (repeated left shifts) and additions.

The general multiplication algorithm, then, consists of a series of shifts and additions, an addition being performed for each '1' bit in the multiplier, indicating a power of two that has to be added.

See AN526 for a flowchart illustrating the process.

Here is the 8-bit unsigned multiplication routine from AN526:

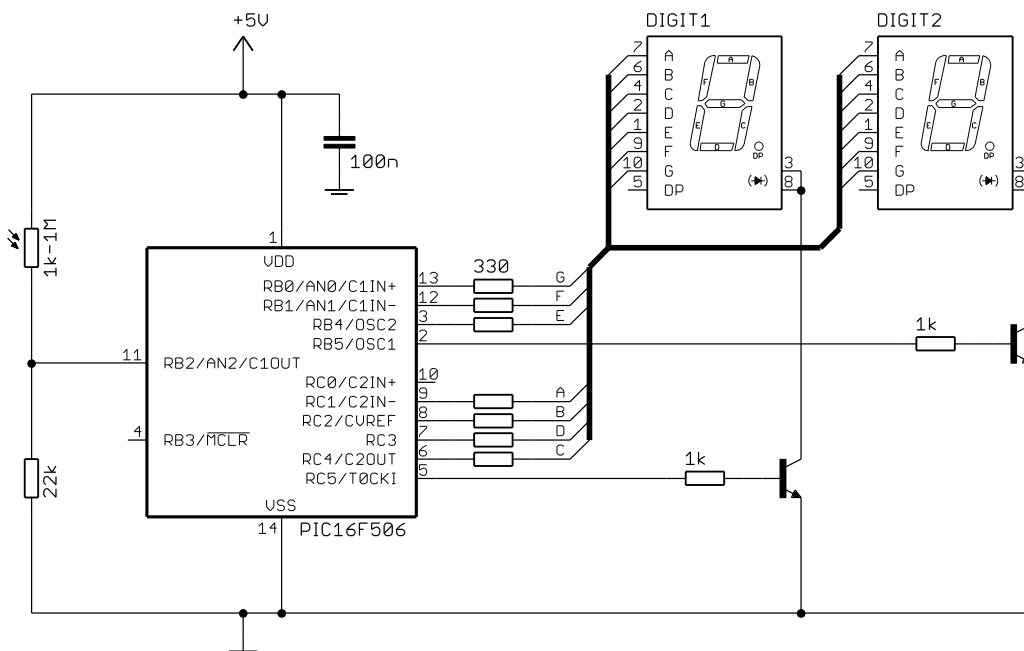
```

; Variables:
; mulcnd - 8 bit multiplicand
; mulplr - 8 bit multiplier
; H_byte - High byte of the 16 bit result
; L_byte - Low byte of the 16 bit result
; count - loop counter
;
; ***** Begin Multiplier Routine
mpy_S   clrf   H_byte           ; start with result = 0
        clrf   L_byte
        movlw  8               ; count = 8
        movwf  count
        movf   mulcnd,w        ; multiplicand in W
        bcf   STATUS,C         ; and carry clear
loop    rrf   mulplr,f         ; right shift multiplier
        btfs  STATUS,C         ; if low-order bit of multiplier was set
        addwf H_byte,f         ; add multiplicand to MSB of result
        rrf   H_byte,f         ; right shift result
        rrf   L_byte,f
        decfsz count,f        ; repeat for all 8 bits
        goto  loop
    
```

It may seem strange that `rrf` is being used here, instead of `rlf`. This is because the multiplicand is being added to the MSB of the result, before being right shifted. The multiplier is processed starting from bit 0. Suppose that bit 0 of the multiplier is a '1'. The multiplicand will be added to the MSB of the result in the first loop iteration. After all eight iterations, it will have been shifted down (right) into the LSB. Subsequent multiplicand additions, corresponding to higher multiplier bits, won't be shifted down as far, so their contribution to the final result is higher. You may need to work an example on paper to see how it works...

Example 1: Light meter with decimal output

[Lesson 10](#) included a simple light meter based on a light-dependent resistor, which displayed the 8-bit ADC output as a two-digit hexadecimal number, using 7-segment LED displays, as shown below:



That's adequate for demonstrating the operation of the ADC module, but it's not a very good light meter. Most people would find it easier to read the display if it was in decimal, not hex, with a scale from 00 – 99 instead of 00h – FFh.

To scale the ADC output from 0 – 255 to 0 – 99, it has to be multiplied by 99/255.

Multiplying by 99 isn't difficult, but dividing by 255 is.

The task is made much easier by using an approximation: instead of multiplying by 99/255, multiply by 100/256. That's a difference of 0.6%; not really significant, given that the ADC is only accurate to ± 2 lsb (2/256, or 0.8%) in any case.

Dividing by 256 is trivial – to divide a 16-bit number by 256, the result is already there – it's simply the most significant byte, with the LSB being the remainder. That gives a result which is always rounded down; if you want to round "correctly", increment the result if the LSB is greater than 127 (LSB<7> = 1). For example:

```
; Variables:
; a = 16-bit value (little endian)
; b = a / 256 (rounded)
    movf    a+1,w          ; result = MSB
    btfsc  a,7            ; if LSB<7> = 1
    incf   a+1,w          ; result = MSB+1
    movwf  b              ; write result
```

Note that, if MSB = 255 and LSB > 127, the result will "round" to zero; probably not what you want.

And in this example, since we're scaling the output to 0 – 99, we wouldn't want to round the result up to 100, since it couldn't be displayed in two digits. We could check for that case and handle it, but it's easiest to simply ignore rounding, and that's valid, because the numbers displayed on the light meter don't correspond to any "real" units which would need to be accurately measured. In other words, the display is in arbitrary units; regardless of the rounding, it will display higher numbers in brighter light, and that's all we're trying to do.

To multiply the raw ADC result by 100, we can adapt the routine from AN526:

```
; scale to 0-99: adc_dec = adc_out * 100
; -> MSB of adc_dec = adc_out * 100 / 256
    clrf   adc_dec          ; start with adc_dec = 0
    clrf   adc_dec+1
    movlw  .8               ; count = 8
    movwf  mpy_cnt
    movlw  .100             ; multiplicand (100) in W
    bcf    STATUS,C        ; and carry clear
l_mpy    rrf    adc_out,f    ; right shift multiplier
    btfsc  STATUS,C        ; if low-order bit of multiplier was set
    addwf  adc_dec+1,f     ; add multiplicand (100) to MSB of result
    rrf    adc_dec+1,f     ; right shift result
    rrf    adc_dec,f
    decfsz mpy_cnt,f       ; repeat for all 8 bits
    goto   l_mpy
```

The 16-bit variable 'adc_dec' now holds the raw ADC result multiplied by 100.

This means that most significant byte of 'adc_dec' (the value stored in the memory location 'adc_dec+1') is equal to the raw ADC result $\times 100/256$.

After scaling the ADC result, we need to extract the “tens” and “ones” digits from it.

That can be done by repeated subtraction; the “tens” digit is determined by continually subtracting 10 from the original value, counting the subtractions until the remainder is less than 10. The “ones” digit is then simply the remainder:

```

        ; extract digits of result
        movf   adc_dec+1,w      ; start with scaled result
        movwf  ones            ;   in ones digit
        clrf   tens            ; and tens clear
l_bcd   movlw  .10              ; subtract 10 from ones
        subwf  ones,w
        btfss STATUS,C        ; (finish if < 10)
        goto  end_bcd
        movwf  ones
        incf   tens,f          ; increment tens
        goto  l_bcd           ; repeat until ones < 10
end_bcd

```

The ‘ones’ and ‘tens’ variables now hold the two digits to be displayed.

Complete program

The rest of the program is essentially the same as the hexadecimal-output example from [lesson 10](#). Here is how the scaling and digit extraction routines fit in:

```

;*****
;
; Description:   Lesson 11, example 1
;
; Displays ADC output in decimal on 2-digit 7-segment LED display
;
; Continuously samples analog input, scales result to 0 - 99
; and displays as 2 x dec digits on multiplexed 7-seg displays
;
;*****
;
; Pin assignments:
; AN2           = voltage to be measured (e.g. pot or LDR)
; RB0-1, RB4, RC1-4 = 7-segment display bus (common cathode)
; RC5           = tens digit enable (active high)
; RB5           = ones digit enable
;
;*****

list          p=16F506
#include      <p16F506.inc>

radix        dec

;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG      _MCLRE_ON & _CP_OFF & _WDT_OFF & _IOSCFS_OFF & _IntRC_OSC_RB4EN

; pin assignments
#define TENS_EN    PORTC,5    ; tens digit enable
#define ONES_EN    PORTB,5    ; ones digit enable

```

```

;***** VARIABLE DEFINITIONS
        UDATA
adc_out  res 1          ; raw ADC output
adc_dec  res 2          ; scaled ADC output (LE 16 bit, 0-99 in MSB)
mpy_cnt  res 1          ; multiplier count
                        ; digits to be displayed:
tens     res 1          ; tens
ones     res 1          ; ones

temp     res 1          ; (temp storage used by set7seg)

;***** RC CALIBRATION
RCCAL    CODE    0x3FF          ; processor reset vector
        res 1          ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET    CODE    0x000          ; effective reset vector
        movwf    OSCCAL          ; apply internal RC factory calibration
        pagesel  start
        goto     start          ; jump to main code

;***** SUBROUTINE VECTORS
set7seg  pagesel  set7seg_R
        goto     set7seg_R      ; display digit on 7-segment display

;***** MAIN PROGRAM *****
MAIN     CODE
;***** Initialisation
start
        ; configure ports
        clrw          ; configure PORTB and PORTC as all outputs
        tris    PORTB
        tris    PORTC
        clrf    CM1CON0          ; disable comparator 1 -> RB0, RB1 digital
        clrf    CM2CON0          ; disable comparator 2 -> RC0, RC1 digital
        clrf    VRCON           ; disable CVref -> RC2 usable

        ; configure ADC
        movlw    b'01111001'      ; configure ADC:
                ; 01-----      AN2 (only) analog (ANS = 01)
                ; --11-----      clock = INTOSC/4 (ADCS = 11)
                ; ----10--        select channel AN2 (CHS = 10)
                ; -----1        turn ADC on (ADON = 1)
        movwf    ADCON0          ; -> AN2 ready for sampling

        ; configure timer
        movlw    b'11010111'      ; configure Timer0:
                ; --0-----      timer mode (T0CS = 0) -> RC5 usable
                ; ----0---        prescaler assigned to Timer0 (PSA = 0)
                ; -----111      prescale = 256 (PS = 111)
        option          ; -> increment every 256 us
                        ; (TMR0<2> cycles every 2.048ms)

;***** Main loop
main_loop
        ; sample input
        bsf     ADCON0,GO          ; start conversion

```



```

w_adc   btfsc   ADCON0,NOT_DONE ; wait until conversion complete
        goto    w_adc
        movf    ADRES,w          ; save ADC result in adc_out
        banksel adc_out
        movwf   adc_out

        ; scale to 0-99: adc_dec = adc_out * 100
        ; -> MSB of adc_dec = adc_out * 100 / 256
        clrf   adc_dec          ; start with adc_dec = 0
        clrf   adc_dec+1
        movlw  .8                ; count = 8
        movwf  mpy_cnt
        movlw  .100              ; multiplicand (100) in W
        bcf    STATUS,C          ; and carry clear
l_mpy   rrf     adc_out,f         ; right shift multiplier
        btfsc  STATUS,C          ; if low-order bit of multiplier was set
        addwf  adc_dec+1,f       ; add multiplicand (100) to MSB of result
        rrf    adc_dec+1,f       ; right shift result
        rrf    adc_dec,f         ; right shift result
        decfsz mpy_cnt,f         ; repeat for all 8 bits
        goto   l_mpy

        ; extract digits of result
        movf   adc_dec+1,w       ; start with scaled result
        movwf  ones              ; in ones digit
        clrf   tens              ; and tens clear
l_bcd   movlw  .10                ; subtract 10 from ones
        subwf  ones,w
        btfss  STATUS,C          ; (finish if < 10)
        goto   end_bcd
        movwf  ones
        incf   tens,f            ; increment tens
        goto   l_bcd
end_bcd

        ; display tens digit for 2.048 ms
w10_hi  btfss  TMR0,2            ; wait for TMR<2> to go high
        goto   w10_hi
        movf   tens,w            ; output tens digit
        pagesel set7seg
        call   set7seg
        pagesel $
        bsf    TENS_EN           ; enable tens display
w10_lo  btfsc  TMR0,2            ; wait for TMR<2> to go low
        goto   w10_lo

        ; display ones digit for 2.048 ms
w1_hi   btfss  TMR0,2            ; wait for TMR<2> to go high
        goto   w1_hi
        banksel ones              ; output ones digit
        movf   ones,w
        pagesel set7seg
        call   set7seg
        pagesel $
        bsf    ONES_EN           ; enable ones display
w1_lo   btfsc  TMR0,2            ; wait for TMR<2> to go low
        goto   w1_lo

        ; repeat forever
        goto   main_loop

```

```

;***** LOOKUP TABLES *****
TABLES CODE 0x200 ; locate at beginning of a page

; pattern table for 7 segment display on port B
; RB4 = E, RB1:0 = FG
get7sB addwf PCL,f
      retlw b'010010' ; 0
      retlw b'000000' ; 1
      retlw b'010001' ; 2
      retlw b'000001' ; 3
      retlw b'000011' ; 4
      retlw b'000011' ; 5
      retlw b'010011' ; 6
      retlw b'000000' ; 7
      retlw b'010011' ; 8
      retlw b'000011' ; 9

; pattern table for 7 segment display on port C
; RC4:1 = CDBA
get7sC addwf PCL,f
      retlw b'011110' ; 0
      retlw b'010100' ; 1
      retlw b'001110' ; 2
      retlw b'011110' ; 3
      retlw b'010100' ; 4
      retlw b'011010' ; 5
      retlw b'011010' ; 6
      retlw b'010110' ; 7
      retlw b'011110' ; 8
      retlw b'011110' ; 9

; Display digit passed in W on 7-segment display
set7seg_R
; disable displays
clrf PORTB ; clear all digit enable lines on PORTB
clrf PORTC ; and PORTC

; output digit pattern
banksel temp
movwf temp ; save digit
call get7sB ; lookup pattern for port B
movwf PORTB ; then output it
movf temp,w ; get digit
call get7sC ; then repeat for port C
movwf PORTC
retlw 0

END

```

Moving Averages, Indirect Addressing and Arrays

Moving averages

A problem with the light meter, as developed so far, is that the display can become unreadable in fluorescent light, because fluorescent lights flicker (too fast for the human eye to notice), and since the meter reacts very quickly (244 samples per second), the display changes too fast to follow.

One solution would be to reduce the sampling rate, to say one sample per second, so that the changes become slow enough for a human to see. But that's not a good solution; the display would still jitter significantly, since some samples would be taken when the illumination was high and others when it was low.

Instead of using a single raw sample, it is often better to smooth the results by implementing a *filter* based on a number of samples over time (a *time series*). Many filter algorithms exist, with various characteristics.

One that is particularly easy to implement is the *simple moving average*, also known as a *box filter*. This is simply the mean value of the last N samples. It is important to average enough samples to produce a smooth result, and to maintain a fast response time, a new average should be calculated every time a new sample is read. For example, you could keep the last ten samples, and then to calculate the simple moving average by adding all the sample values and then dividing by ten. Whenever a new sample is read, it is added to the list, the oldest sample is discarded, and the calculation is repeated. In fact, it is not necessary to repeat all the additions; it is only necessary to subtract the oldest value (the sample being discarded) and to add the new sample value.

Sometimes it makes more sense to give additional weight to more recent samples, so that the moving average more closely tracks the most recent input. A number of forms of *weighting* can be used, including arithmetic and exponential, which require more calculation. But a simple moving average is sufficient for our purpose here.

Indirect addressing and arrays

Instead of talking about a “list” of samples, we'd normally call it an *array*.

An array is a contiguous set of variables which can be accessed through a numeric index.

For example, to calculate an average in C, you might write something like:

```
int s[10];          /* array of samples */
int avg;           /* sample average */
int i;

avg = 0;
for (i = 0; i < 10; i++) /* add all the samples */
    avg = avg + s[i];
avg = avg / 10;      /* divide by 10 to calculate average */
```

But how could we do that in PIC assembler?

You could define a series of variables: s0, s1, s2, ... , s9, but there is then no way to add them in a loop, since each variable would have to be referred to by its own block of code. That would make for a long, and difficult to maintain program.

There is of course a way: the baseline PICs support *indirect addressing* (making array indexing possible), through the FSR and INDF registers.

The INDF (**indirect file**) “register” acts as a window, through which the contents of any other register can be accessed.

The FSR (**file select register**) holds the address of the register which will be accessed through INDF.

For example, if FSR = 08h, INDF accesses the register at address 08h, which is CM1CON0 on the PIC16F506.

So, on the PIC16F506, if FSR = 08h, reading or writing INDF is the same as reading or writing CM1CON0.

Recall that the bank selection bits form the upper bits of the FSR register.

When you write a value into FSR, INDF will access the register at the address given by that value, irrespective of banking. That is, indirect addressing allows linear, un-banked access to the register file.

For example, if FSR = 54h, INDF will access the register at address 54h; this happens to be in bank 2, but that's not a consideration when using indirect addressing.

Note: When FSR is updated for indirect register access, the bank selection bits will be overwritten.

The PIC12F510/16F506 data sheet includes the following code to clear registers 10h – 1Fh:

```

        movlw    0x10        ; initialize pointer to RAM
        movwf   FSR
next    clrf    INDF        ; indirectly clear register (pointed to by FSR)
        incf   FSR,f        ; inc pointer
        btfsc  FSR,4        ; all done?
        goto   next        ; NO, clear next
continue
                                ; YES, continue

```

The 'clrf INDF' instruction clears the register pointed to by FSR, which is incremented from 10h to 1Fh.

Note that at the test at the end of the loop, 'btfsc FSR,4', finishes the loop when the end of bank 0 (1Fh) has been reached. In fact, this test can be used for the end of any bank, not just bank 0.

Example 2: Light meter with smoothed decimal output

To effectively smooth the light meter's output, so that it doesn't jitter under fluorescent lighting, a simple moving average is quite adequate – assuming that the sample *window* (the time that samples are averaged over) is longer than the variations to be smoothed.

The electricity supply, and hence the output of most A/C lighting, cycles at 50 or 60 Hz in most places. A 50 Hz cycle is 20 ms long, so the sample window needs to be longer than that. Our example light meter program samples every 4 ms, so at least five samples need to be averaged ($5 \times 4 \text{ ms} = 20 \text{ ms}$) to smooth a 50 Hz cycle. But a longer window would be better; two or three times the cycle time would ensure that cyclic variations are smoothed out.

We have seen that the data memory on any baseline PIC with multiple data memory banks is not contiguous. The 16F506 has four banked 16-byte general purpose register (GPR) regions (forming the “top half” of each of the four banks), plus one 3-byte non-banked (or shared) GPR region. Thus, the largest contiguous block of memory that can be allocated on the 16F506 is 16 bytes. It is easiest to implement arrays if they are contiguous, so the largest single array we can easily define is 16 bytes – which happens to be a good size for the sample array (or *buffer*) for this application.

Since each data section has to fit within a single data memory region, and the largest available data memory region on a PIC16F506 is 16 bytes, if you try something like:

```

        UDATA
adc_dec  res 2                ; scaled ADC output (LE 16 bit, 0-99 in MSB)
mpy_cnt  res 1                ; multiplier count
smp_buf  res 16              ; array of samples for moving average

```

you will get a “'.udata' can not fit the section” error from the linker, because we have tried to reserve a total of 19 bytes in a single UDATA section. Unnamed UDATA sections are given the default name '.udata', so the error message is telling us that this section, which is named '.udata', is too big.

So we need to split the variable definitions into two (or more) UDATA sections, with no more than 16 bytes in each section. To declare more than one UDATA section, they have to have different names, for example:

```

VAR1    UDATA
adc_dec res 2           ; scaled ADC output (LE 16 bit, 0-99 in MSB)
mpy_cnt res 1           ; multiplier count

ARRAY1  UDATA
smp_buf res 16          ; array of samples for moving average

```

Although we don't know which bank the array will be placed in, we do know that it will fill the whole of one of the 16-byte banked GPR memory regions, forming the top half of whichever bank it is in.

That means that to clear the array, we can adapt the code from the data sheet:

```

        ; clear sample buffer
        movlw  smp_buf
        movwf  FSR
l_clr   clrf   INDF           ; clear each byte
        incf   FSR, f
        btfsc  FSR, 4         ; until end of bank is reached
        goto   l_clr

```

This approach wouldn't work if the array was any smaller than 16 bytes, in which case we would need to use a subtraction or XOR to test for FSR reaching the end of the array.

Since the 16-byte array uses all the banked data space in one bank, there is no additional room in that bank to store any other variables we may need to access while working with the array, such as the running total of sample values in the array. In the baseline architecture, accessing variables in other banks is very awkward when using indirect memory access, because selecting another bank means changing FSR, which is being used to access the array.

To reduce the number of bank selection changes necessary, and the need to save/restore FSR after each one, it makes sense to place variables associated with the array in shared memory, wherever possible.

For example:

```

SHR1    UDATA_SHR
adc_sum res 2           ; sum of samples (LE 16-bit), for average
adc_avg res 1           ; average ADC output

```

It was ok to work directly with FSR in the "clear sample buffer" loop above, since it is short and no bank selection occurs within it. But it's not practical to remove the need for banking altogether throughout the sampling loop, where we read a sample, update the moving average calculation, scale the result, convert it to decimal and then display it, before moving on to the next sample. So we need to save the pointer to the "current" sample in a variable ('smp_idx') which will not be overwritten when a bank is selected.

Updating and calculating the total of the samples (stored in a 16-bit variable called 'adc_sum') is done as follows:

```

        banksel smp_idx
        movf   smp_idx, w      ; set FSR to current sample buffer index
        movwf  FSR
        movf   INDF, w         ; subtract old sample from running total
        subwf  adc_sum, f
        btfss  STATUS, C
        decf   adc_sum+1, f
        movf   ADRES, w        ; save new sample (ADC result)
        movwf  INDF

```

```

addwf   adc_sum, f           ; and add to running total
btfsc   STATUS, C
incf    adc_sum+1, f

```

This total then has to be divided by 16 (the number of samples) to give the moving average.

As we've seen, dividing by any power of two can be simply done through a series of right-shifts. In this case, since we need to keep 'adc_sum' intact from one loop iteration to the next (to maintain the running total), we would need to take a copy of it and right-shift the copy four times (to divide by 16). Since 'adc_sum' is a 16-bit quantity, both the MSB and LSB would have to be right-shifted, so we'd need eight right-shifts in total, plus a few instructions to copy 'adc_sum' – around a dozen instructions in total.

But since we need to right-shift by four bits, and the `swapf` instruction swaps the nybbles (four bits) in a byte, shifting the upper nybble right by four bits, we can use it to divide by 16 more efficiently.

Suppose the running total in 'adc_sum' is 0ABCh. (The upper nybble will always be zero because the result of adding 16 eight-bit numbers is a twelve-bit number; the sum can never be more than 0FF0h).

The result we want (0ABCh divided by 16, or right-shifted four times) is ABh.

Swapping the nybbles in the LSB gives CBh. Next we need to clear the high nybble to remove the 'C', which as we saw in lesson 8, can be done through a masking operation, using AND, leaving 0Bh.

Swapping the nybbles in the MSB gives A0h.

Finally we need to combine the upper nybble in the MSB (A0h) with the lower nybble in the LSB (0Bh).

This can be done with an inclusive-or, since any bit ORed with '0' remains unchanged, while any bit ORed with '1' is set to '1'. That is:

$$n \text{ OR } 0 = n$$

$$n \text{ OR } 1 = 1$$

So, for example, A0h OR 0Bh = ABh. (In binary, 1010 0000 OR 0000 1011 = 1010 1011.)

The baseline PICs provide two "inclusive-or" instructions:

`iorwf` – "inclusive-or **W** with register file"

`iorlw` – "inclusive-or literal with **W**"

These are used in the same way as the exclusive-or instructions we've seen before.

For completeness, the baseline PICs provide one more logic instruction we haven't covered so far:

`andwf` – "and **W** with register file"

We can use 'swapf' to rearrange the nybbles, 'andlw' to mask off the unwanted nybble, and 'iorwf' to combine the bytes, creating an efficient "divide by 16" routine, as follows:

```

swapf   adc_sum, w           ; divide total by 16
andlw   0x0F
movwf   adc_avg
swapf   adc_sum+1, w
iorwf   adc_avg, f

```

The result is the moving average, which can be scaled, converted to decimal and displayed as before.

Complete program

Although much of this code is the same as in the previous example, here is the complete “light meter with smoothed decimal display” program, showing how all the parts fit together:

```

;*****
;
; Description: Lesson 11, example 2
;
; Demonstrates use of indirect addressing
; to implement a simple moving average filter
;
; Displays ADC output in decimal on 2-digit 7-segment LED display
;
; Continuously samples analog input, averages last 16 samples,
; scales result to 0 - 99 and displays as 2 x dec digits
; on multiplexed 7-seg displays
;
;*****
;
; Pin assignments:
; AN2 = voltage to be measured (e.g. pot or LDR)
; RB0-1, RB4, RC1-4 = 7-segment display bus (common cathode)
; RC5 = tens digit enable (active high)
; RB5 = ones digit enable
;
;*****

list p=16F506
#include <p16F506.inc>

radix dec

;***** CONFIGURATION
; ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG _MCLRE_ON & _CP_OFF & _WDT_OFF & _IOSCFE_OFF & _IntRC_OSC_RB4EN

; pin assignments
#define TENS_EN PORTC,5 ; tens digit enable
#define ONES_EN PORTB,5 ; ones digit enable

;***** VARIABLE DEFINITIONS
VAR1 UDATA
adc_dec res 2 ; scaled ADC output (LE 16 bit, 0-99 in MSB)
mpy_cnt res 1 ; multiplier count
smp_idx res 1 ; index into sample array
; digits to be displayed:
tens res 1 ; tens
ones res 1 ; ones

temp res 1 ; (temp storage used by set7seg)

ARRAY1 UDATA
smp_buf res 16 ; array of samples for moving average

SHR1 UDATA_SHR
adc_sum res 2 ; sum of samples (LE 16-bit), for average
adc_avg res 1 ; average ADC output

```

```

;***** RC CALIBRATION
RCCAL   CODE    0x3FF           ; processor reset vector
        res 1                 ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET   CODE    0x000           ; effective reset vector
        movwf OSCCAL          ; apply internal RC factory calibration
        pagesel start
        goto    start          ; jump to main code

;***** SUBROUTINE VECTORS
set7seg           ; display digit on 7-segment display
        pagesel set7seg_R
        goto    set7seg_R

;***** MAIN PROGRAM *****
MAIN     CODE

;***** Initialisation
start
        ; configure ports
        clrw                    ; configure PORTB and PORTC as all outputs
        tris    PORTB
        tris    PORTC
        clrf    CM1CON0         ; disable comparator 1 -> RB0, RB1 digital
        clrf    CM2CON0         ; disable comparator 2 -> RC0, RC1 digital
        clrf    VRCON           ; disable CVref -> RC2 usable

        ; configure ADC
        movlw   b'01111001'     ; configure ADC:
                                ; 01----- AN2 (only) analog (ANS = 01)
                                ; --11---- clock = INTOSC/4 (ADCS = 11)
                                ; ----10-- select channel AN2 (CHS = 10)
                                ; -----1 turn ADC on (ADON = 1)
        movwf   ADCON0          ; -> AN2 ready for sampling

        ; configure timer
        movlw   b'11010111'     ; configure Timer0:
                                ; --0----- timer mode (T0CS = 0) -> RC5 usable
                                ; ----0--- prescaler assigned to Timer0 (PSA = 0)
                                ; -----111 prescale = 256 (PS = 111)
        option           ; -> increment every 256 us
                                ; (TMR0<2> cycles every 2.048ms)

        ; initialise variables
        clrf    adc_sum         ; sample buffer total = 0
        clrf    adc_sum+1

        ; clear sample buffer
        movlw   smp_buf
        movwf   FSR
l_clr    clrf    INDF           ; clear each byte
        incf    FSR,f
        btfsc   FSR,4           ; until end of bank is reached
        goto    l_clr

;***** Main loop
main_loop

```



```

        ; set index to start of sample buffer
        movlw   smp_buf
        banksel smp_idx
        movwf   smp_idx

; *** repeat for each sample in buffer
l_smp_buf

        ; sample input
        bsf     ADCON0,GO           ; start conversion
w_adc   btfsc   ADCON0,NOT_DONE    ; wait until conversion complete
        goto    w_adc

        ; calculate moving average
        banksel smp_idx
        movf    smp_idx,w          ; set FSR to current sample buffer index
        movwf   FSR
        movf    INDF,w             ; subtract old sample from running total
        subwf   adc_sum,f
        btfss   STATUS,C
        decf    adc_sum+1,f
        movf    ADRES,w           ; save new sample (ADC result)
        movwf   INDF
        addwf   adc_sum,f         ; and add to running total
        btfsc   STATUS,C
        incf    adc_sum+1,f
        swapf   adc_sum,w         ; divide total by 16
        andlw   0x0F
        movwf   adc_avg
        swapf   adc_sum+1,w
        iorwf   adc_avg,f

        ; scale to 0-99: adc_dec = adc_avg * 100
        ; -> MSB of adc_dec = adc_avg * 100 / 256
        banksel adc_dec
        clrf    adc_dec           ; start with adc_dec = 0
        clrf    adc_dec+1
        movlw   .8                ; count = 8
        movwf   mpy_cnt
        movlw   .100              ; multiplicand (100) in W
        bcf     STATUS,C          ; and carry clear
l_mpy   rrf     adc_avg,f         ; right shift multiplier
        btfsc   STATUS,C          ; if low-order bit of multiplier was set
        addwf   adc_dec+1,f       ; add multiplicand (100) to MSB of result
        rrf     adc_dec+1,f       ; right shift result
        rrf     adc_dec,f
        decfsz  mpy_cnt,f         ; repeat for all 8 bits
        goto    l_mpy

        ; extract digits of result
        movf    adc_dec+1,w       ; start with scaled result
        movwf   ones              ; in ones digit
        clrf    tens              ; and tens clear
l_bcd   movlw   .10               ; subtract 10 from ones
        subwf   ones,w
        btfss   STATUS,C          ; (finish if < 10)
        goto    end_bcd
        movwf   ones
        incf    tens,f            ; increment tens
        goto    l_bcd            ; repeat until ones < 10
end_bcd

```

```

; display tens digit for 2.048 ms
w10_hi  btfss   TMR0,2           ; wait for TMR<2> to go high
        goto    w10_hi
        movf    tens,w           ; output tens digit
        pagesel set7seg
        call    set7seg
        pagesel $
w10_lo  btfsc   TMR0,2           ; wait for TMR<2> to go low
        goto    w10_lo

; display ones digit for 2.048 ms
w1_hi   btfss   TMR0,2           ; wait for TMR<2> to go high
        goto    w1_hi
        banksel ones            ; output ones digit
        movf    ones,w
        pagesel set7seg
        call    set7seg
        pagesel $
w1_lo   btfsc   TMR0,2           ; wait for TMR<2> to go low
        goto    w1_lo

; end sample buffer loop
banksel smp_idx           ; increment sample buffer index
incf    smp_idx,f
btfsc   smp_idx,4         ; repeat loop until end of buffer
goto    l_smp_buf

; repeat main loop forever
goto    main_loop

;***** LOOKUP TABLES
TABLES  CODE      0x200           ; locate at beginning of a page

; pattern table for 7 segment display on port B
; RB4 = E, RB1:0 = FG
get7sB  addwf    PCL,f
        retlw   b'010010'         ; 0
        retlw   b'000000'         ; 1
        retlw   b'010001'         ; 2
        retlw   b'000001'         ; 3
        retlw   b'000011'         ; 4
        retlw   b'000011'         ; 5
        retlw   b'010011'         ; 6
        retlw   b'000000'         ; 7
        retlw   b'010011'         ; 8
        retlw   b'000011'         ; 9

; pattern table for 7 segment display on port C
; RC4:1 = CDBA
get7sC  addwf    PCL,f
        retlw   b'011110'         ; 0
        retlw   b'010100'         ; 1
        retlw   b'001110'         ; 2
        retlw   b'011110'         ; 3
        retlw   b'010100'         ; 4
        retlw   b'011010'         ; 5
        retlw   b'011010'         ; 6
        retlw   b'010110'         ; 7

```

```

        retlw    b'011110'        ; 8
        retlw    b'011110'        ; 9

; Display digit passed in W on 7-segment display
set7seg_R
    ; disable displays
    clrf    PORTB                ; clear all digit enable lines on PORTB
    clrf    PORTC                ; and PORTC

    ; output digit pattern
    banksel temp
    movwf   temp                ; save digit
    call    get7sB              ; lookup pattern for port B
    movwf   PORTB               ; then output it
    movf    temp,w              ; get digit
    call    get7sC              ; then repeat for port C
    movwf   PORTC
    retlw   0

END

```

You should find that the resulting display is stable, even under fluorescent lighting, and yet still responds quickly to changing light levels.

Conclusion

This tutorial series has now introduced every baseline PIC instruction and every special function register (except those associated with EEPROM access on those few baseline PICs with EEPROMs).

That concludes our introduction to the baseline PIC architecture and assembly programming.

The material in these lessons is revisited in a tutorial series on [programming baseline PICs in C](#).

In that series it becomes apparent that some tasks are more easily expressed in C than assembler, especially the most recent topic of arithmetic and arrays, but that C can be relatively inefficient. It is also seen that different C compilers take different approaches – with pros and cons that become apparent as the various examples are implemented in each.

Now that you have a basic understanding of programming baseline PICs in assembler (and C, if you go through the [baseline C tutorial series](#)), you may wish to move on to the [midrange PIC architecture and assembler tutorials](#), where you will be introduced to the more flexible and capable midrange PIC core, and some of its diverse range of peripherals. These lessons are also followed up by a series on [programming midrange PICs in C](#).

Enjoy!

Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 1: Basic Digital Output

Although assembly language is commonly used to programming small microcontrollers, it is less appropriate for complex applications on larger MCUs; it can become unwieldy and difficult to maintain as programs grow longer. A number of higher-level languages are used in embedded systems development, including BASIC, Forth and even Pascal. But the most commonly used “high level” language is C.

C is often considered to be inappropriate for very small MCUs, such as the baseline PICs we have examined in [the baseline assembler tutorial series](#), because they have limited resources and their architecture is not well suited to C code. However, as this tutorial series will demonstrate, it is quite possible to use C for simple programs on baseline PICs – although it is true that C may not be able to make the most efficient use of the limited memory on these small devices, as we will see in [later lessons](#).

This lesson introduces the “free” Custom Computer Services (CCS) compiler bundled with MPLAB¹, and Microchip’s XC8 compiler (running in “Free mode”), both of which fully support all current baseline PICs². As we’ll see, the XC8 and CCS compilers take quite different approaches to many implementation tasks. Most other PIC C compilers take a similar approach to one or the other, or fall somewhere in between, making these compilers a good choice for an introduction to programming PICs in C.

This lesson covers basic digital output, through that standby of introductory microcontroller courses: flashing LEDs – although the concepts can be applied to anything which can be controlled by a high/low, on/off signal, including power MOSFET switches and relays.

It is assumed that you are already familiar with the material covered in those baseline assembler lessons [1](#) to [3](#). If not, you should review those lessons while working through this one. Specifically, this lesson **does not** provide a detailed overview of the baseline PIC architecture, installing and using MPLAB or programmers such as the PICKit 2. Instead, this lesson explains how to create C projects in MPLAB, and how to implement the examples from the assembler lessons, in C.

In summary, this lesson covers:

- Introduction to the Microchip XC8 and CCS PCB compilers
- Using MPLAB 8 and MPLAB X to create C projects
- Simple control of digital output pins
- Programmed delays

with examples for both compilers.

This tutorial assumes a working knowledge of the C language; it does **not** attempt to teach C.

¹ CCS PCB is bundled with MPLAB 8 only.

² at the time of writing (September 2012)

Introducing XC8 and CCS PCB

Up until version 8.10, MPLAB was bundled with HI-TECH's "PICC-Lite" compiler, which supported all the baseline (12-bit) PICs available at that time, including those used in this tutorial series, with no restrictions. It also supports a small number of the mid-range (14-bit) PICs – although, for most of the mid-range devices it supported, PICC-Lite limited the amount of data and program memory that could be used, to provide an incentive to buy the full compiler. Microchip have since acquired HI-TECH Software, and no longer supply or support PICC-Lite. As such, PICC-Lite will not be covered in these tutorials.

Microchip have developed the former HI-TECH C compiler into their own "MPLAB XC8" compiler. It is available for download at www.microchip.com.

XC8's "Free mode" supports all 8-bit (including baseline and mid-range) PICs, with no memory restrictions. However, in this mode, most compiler optimisation is turned off, making the generated code around twice the size of that generated by PICC-Lite.

This gives those developing for baseline and mid-range PICs easy access to a free compiler supporting a much wider range of devices than PICC-Lite, without memory usage restrictions, albeit at the cost of much larger generated code. And XC8 will continue to be maintained, supporting new baseline and mid-range devices over time.

But if you are using Windows and developing code for a supported baseline PIC, it is quite valid to continue to use PICC-Lite (if you are able to locate a copy – by downloading MPLAB 8.10 from the archives on www.microchip.com, for example), since it will generate much more efficient code, while allowing all the (limited) memory on your baseline PIC to be used. It can be installed alongside XC8. But to repeat – PICC-Lite won't be described in these lessons.

MPLAB 8 includes a free copy of CCS's PCB C compiler for Windows, which supports most baseline PICs, including those used in these tutorials. Although it's now a little date (at the time of writing, the version bundled with MPLAB was 4.073, while the latest commercially available version was 4.135), it remains useful and so is used in these tutorials.

If you are using MPLAB 8, you should select the CCS compiler as an option when installing MPLAB, to ensure that the integration with the MPLAB IDE will be done correctly.

The XC8 installer (for Windows, Linux or Mac) has to be downloaded separately from www.microchip.com. When you run the XC8 installer, you will be asked to enter a license activation key. Unless you have purchased the commercial version, you should leave this blank. You can then choose whether to run the compiler in "Free mode", or activate an evaluation license. We'll be using "Free mode" in these lessons, but it's ok to use the evaluation license (for 60 days) if you choose to.

Custom Computer Services (CCS) "PCB"

CCS (www.ccsinfo.com) specialises in PIC development, offering hardware development platforms, as well as a range of C compilers supporting (as of February 2012) almost all the PIC processors from the baseline 10Fs through to the 16-bit PIC24Fs and dsPICs. They also offer an IDE, including a "C-aware" editor, and debugger/simulator.

"PCB" is the command-line compiler supporting the baseline (12-bit) PICs.

A separate command-line compiler, called "PCM", supports the mid-range (14-bit) PICs, including most PIC16s. Similarly, "PCH" supports the 16-bit instruction-width, 8-bit data width PIC18 series, while "PCD" supports the 24-bit instruction-width, 16-bit data width PIC24 and dsPIC series. These command-line compilers are available for both Windows and Linux. A plug-in allows these compilers to be integrated into both MPLAB 8 and MPLAB X³.

³ Note that the free version of the CCS PCB compiler, supplied with MPLAB 8, **will not** work with MPLAB X.

CCS also offer a Windows IDE, called “PCW”, which incorporates the PCB and PCM compilers. “PCWH” extends this to include PCH for 18F support, while “PCWHD” supports the full suite of PICs. A lower-cost IDE, called “PCDIDE” includes only the PCD compiler.

The CCS compilers and IDEs are relatively inexpensive: as of February 2012, the advertised costs range from US\$50 for PCB, through US\$150 for PCM, US\$350 for PCW, to US\$600 for the full PCWHD suite.

As we’ll see, the CCS approach is to provide a large number of PIC-specific inbuilt functions, such as `read_adc()`, which make it easy to access or use PIC features, without having to be aware of and specify all the registers and bits involved. That means that the CCS compilers can be used without needing a deep understanding of the underlying hardware, which can be a two-edged sword; it is easier to get started and less-error prone (in that the compiler can be expected to set up the registers correctly), but can be less flexible and more difficult to debug when something is wrong (especially if the bug is in the compiler’s implementation, and not your code).

Microchip “MPLAB XC8”

XC8 supports the whole 8-bit PIC10/12/16/18 series in a single edition, with different licence keys unlocking different levels of code optimisation – “Free” (free, but no optimisation), “Standard” and “PRO” (most expensive and highest optimisation).

Microchip XC compilers are also available for the PIC24, dsPIC and PIC32 families.

The XC8 compiler is more expensive than those from CCS: as of August 2012, the advertised costs include US\$495 for the “Standard” mode, and US\$995 for this compiler in “PRO” mode.

We’ll see that XC8 exposes the PIC’s registers as variables, to be accessed “directly” by the developer, in much the same way that they would be in assembler, instead of via built-in functions. This means that, to effectively use the XC8 compiler, you need a strong understanding of the underlying PIC hardware, equivalent to that needed for programming in assembler.

These differing approaches are highlighted in the examples below. Instead of trying to force either compiler into a particular style, the examples for each compiler are written in a style similar in “spirit” to the sample code provided with each. Although it is possible to map registers into variables in the CCS compilers, the examples in these tutorials use the CCS built-in functions where that seems reasonable, since that is how that compiler was intended to be used. However, identical comments are used where reasonable, to highlight the correspondence between both C compilers and the original assembler version of each example.

Data Types

One of the problems with implementing ANSI-standard C on microcontrollers is that there is often a need to work with individual bits, while the smallest data-type included in the ANSI standard is ‘char’, which is normally considered to be a single byte, or 8 bits. Another problem is the length of a standard integer (‘int’) is not defined, being implementation-dependent. Whether an ‘int’ is 16 or 32 bits is an issue on larger systems, but it makes a much more significant difference to code portability on microcontrollers. Similarly, the sizes of ‘float’, ‘double’, and the effect of the modifiers ‘short’ and ‘long’ is not defined by the standard.

So different compilers use various sizes for the “standard” data types, and for microcontroller implementations it is common to add a single-bit type as well – generally specific to that compiler.

Here are the data types and sizes supported by CCS PCB and XC8:

Type	XC8	CCS PCB
bit	1	-
int1	-	1
char	8	8
int8	-	8
short	16	1
int	16	8
int16	-	16
short long	24	-
long	32	16
int32	-	32
float	24 or 32	32
double	24 or 32	-

You'll see that very few of these line up; the only point of agreement is that 'char' is 8 bits!

XC8 defines a single 'bit' type, unique to XC8.

CCS PCB defines 'int1', 'int8', 'int16' and 'int32' types, which make it easy to be explicit about the size of a data element (such as a variable).

The "standard" 'int' type is 8 bits in CCS PCB, but 16 bits in XC8.

But by far the greatest difference is in the definition of 'short': in XC8, it is a synonym for 'int' and is a 16-bit type, whereas in CCS PCB, 'short' is a single-bit type, the same as an 'int1'. That could be very confusing when porting code from CCS PCB to another compiler, so for clarity it is probably best to use 'int1' when defining single-bit variables.

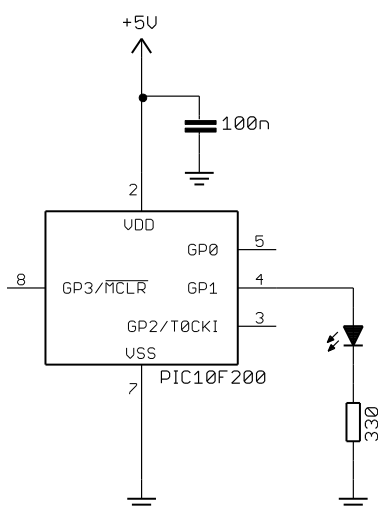
XC8 also offers the non-standard 24-bit 'short long' type. And note that floating-point variables in XC8 can be either 24 or 32 bits; this is set by a compiler option. The only floating-point representation available in CCS PCB is 32-bit, which may be a higher level of precision than is needed in most applications for small applications, so XC8's ability to work with 24-bit floating point numbers can be useful.

To make it easier to create portable code, XC8 provides the 'stdint.h' header file, which defines the C99 standard types such as 'uint8_t' and 'int16_t'.

Unfortunately, CCS PCB does not come with a version of 'stdint.h', although the size of CCS types such as 'int8', and 'int16' is clear.

Example 1: Turning on an LED

In [baseline assembler lesson 1](#) we saw how to turn on a single LED, and leave it on; the (very simple) circuit, using a PIC10F200, is shown below:



This circuit is intended for use with the [Gooligum baseline training board](#), where you can simply plug the 10F200 into the '10F' socket, and connect jumper JP12.

If you have the Microchip Low Pin Count Demo board, which does not support PIC10F devices, you will have to substitute a 12F508 or 12F509 and connect an LED to pin GP1: see [baseline assembler lesson 1](#) for details.

To turn on the LED on GP1, we must clear bit 1 of the TRIS, configuring GP1 as an output, and then set bit 1 of GPIO, setting GP1 high, turning the LED on.

At the start of the assembly language program, the PIC was configured, and the internal RC oscillator was calibrated, by loading the factory calibration value into the OSCCAL register.

Finally, the end of the program consisted of an infinite loop, to leave the LED turned on.

Here are the key parts of the 10F200 version of the assembler code from baseline lesson 1:

```

                ; ext reset, no code protect, no watchdog
__CONFIG      _MCLRE_ON & _CP_OFF & _WDT_OFF

RESET        CODE    0x000                ; effective reset vector
              movwf  OSCCAL                ; apply internal RC factory calibration

              movlw  b'111101'            ; configure GP1 (only) as an output
              tris   GPIO
              movlw  b'000010'            ; set GP1 high
              movwf  GPIO

              goto   $                    ; loop forever

```

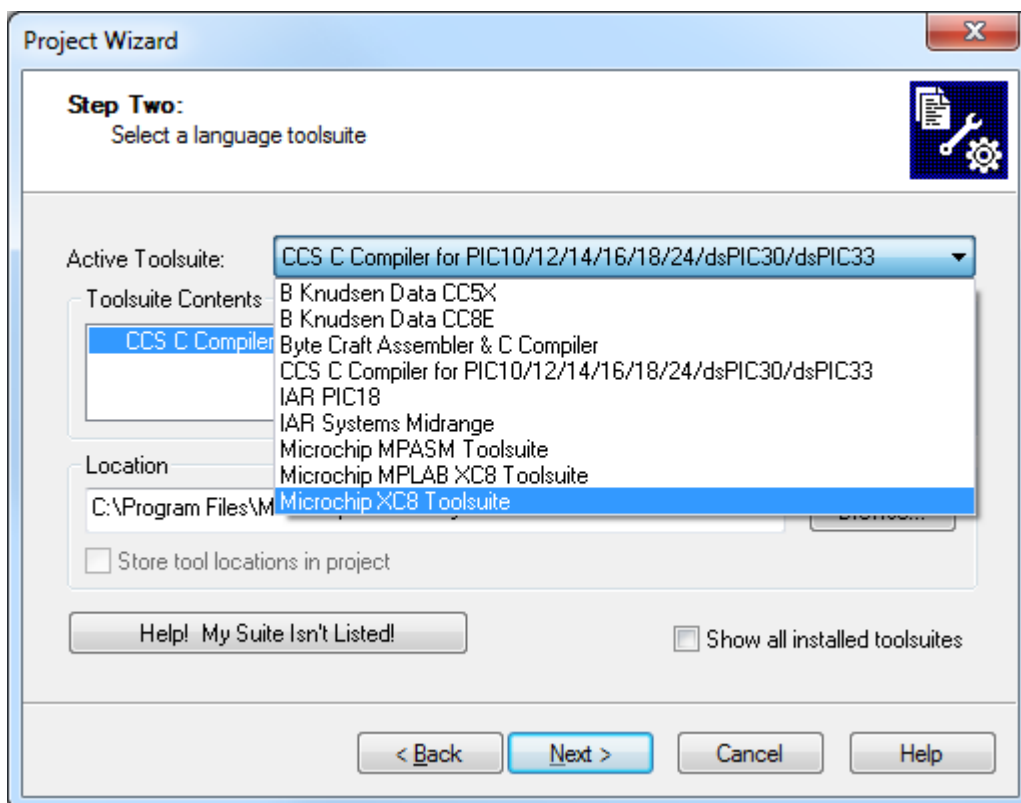
XC8

XC8 projects in MPLAB are created in a similar way to assembler projects, but as we saw in baseline lesson 1, the details depend on which version of the MPLAB IDE you are using.

MPLAB 8.x

You should use the project wizard to create a new project, as before.

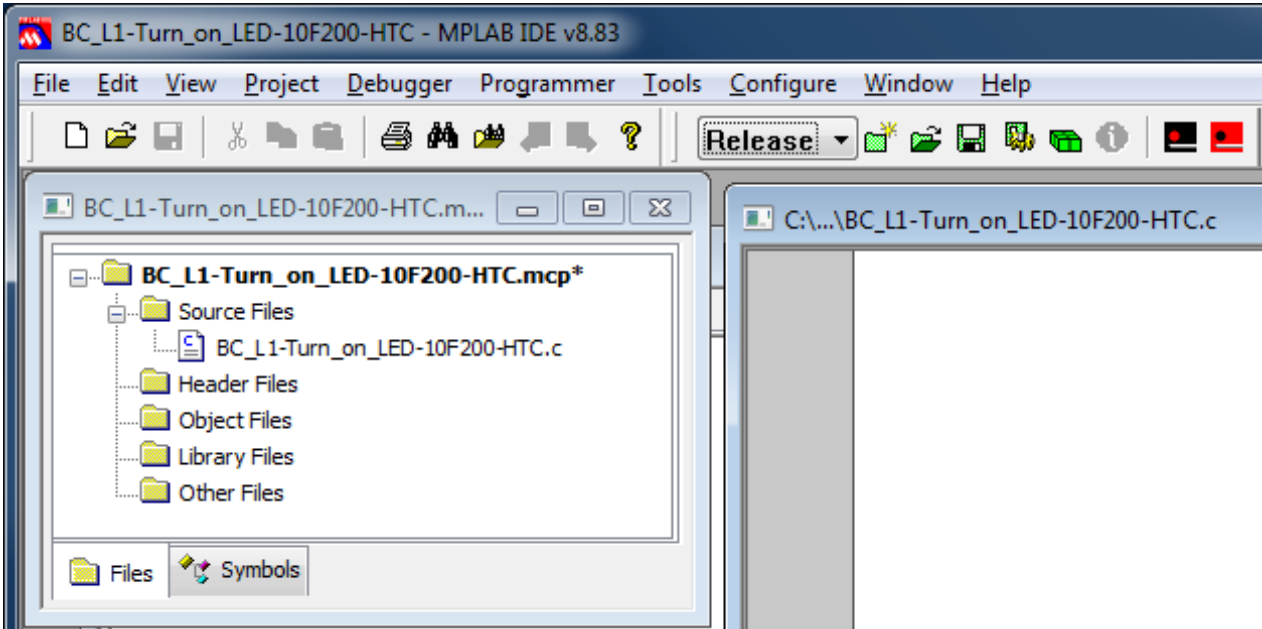
To specify that this is an XC8 project, select the “Microchip XC8 ToolSuite” when you reach “Step Two: Select a language toolsuite”:



When completing the wizard, note that there is no need to add existing files to your project, unless you wish to make use of some existing code, perhaps from a previous project. There is no equivalent to the MPASM “template files” we saw in baseline lesson 1.

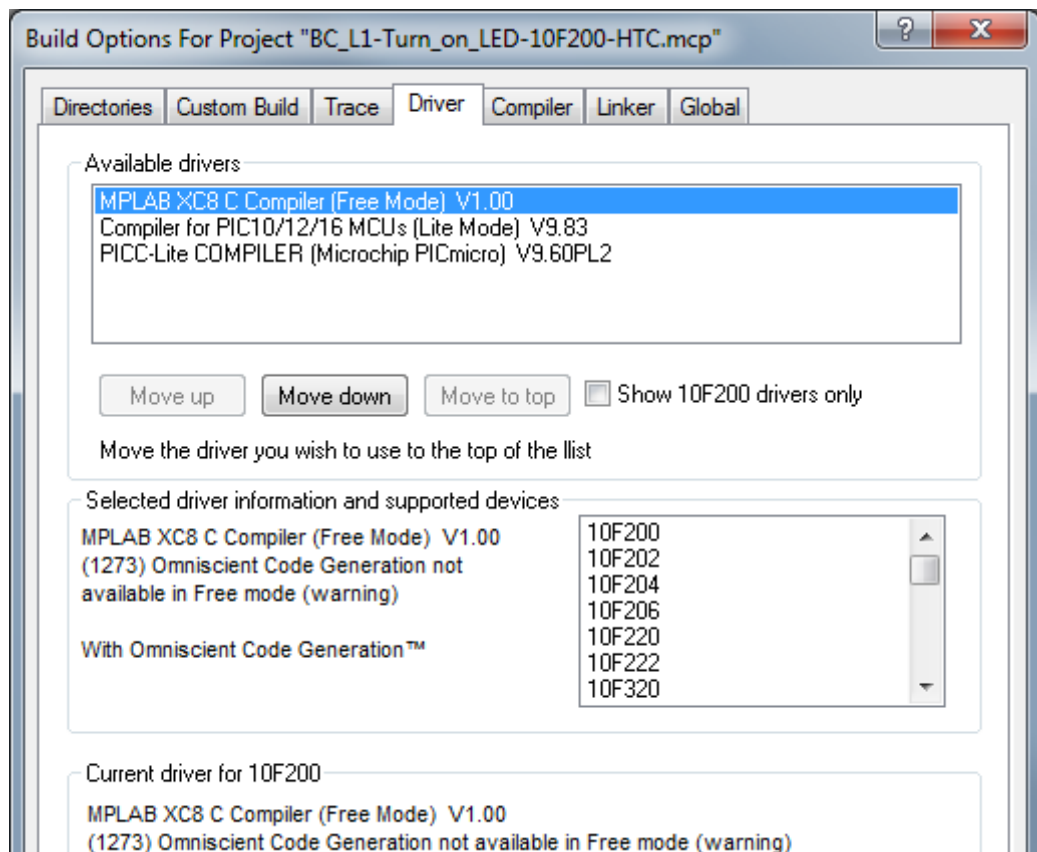
After finishing the project wizard, you can use the “File → Add New File to Project...” menu item to create a ‘.c’ source file in your project folder.

It should appear under “Source Files” in the project window, as usual:



If you have installed more than one HI-TECH or XC8 compiler, you need to tell the XC8 toolsuite which compiler, or driver, to use.

Open the project build options window (Project → Build Options... → Project) then select the “Driver” tab, as shown on the right.



To select the compiler you wish to use, move it to the top of the list of available drivers, by using the “Move up” button). The “Current driver”

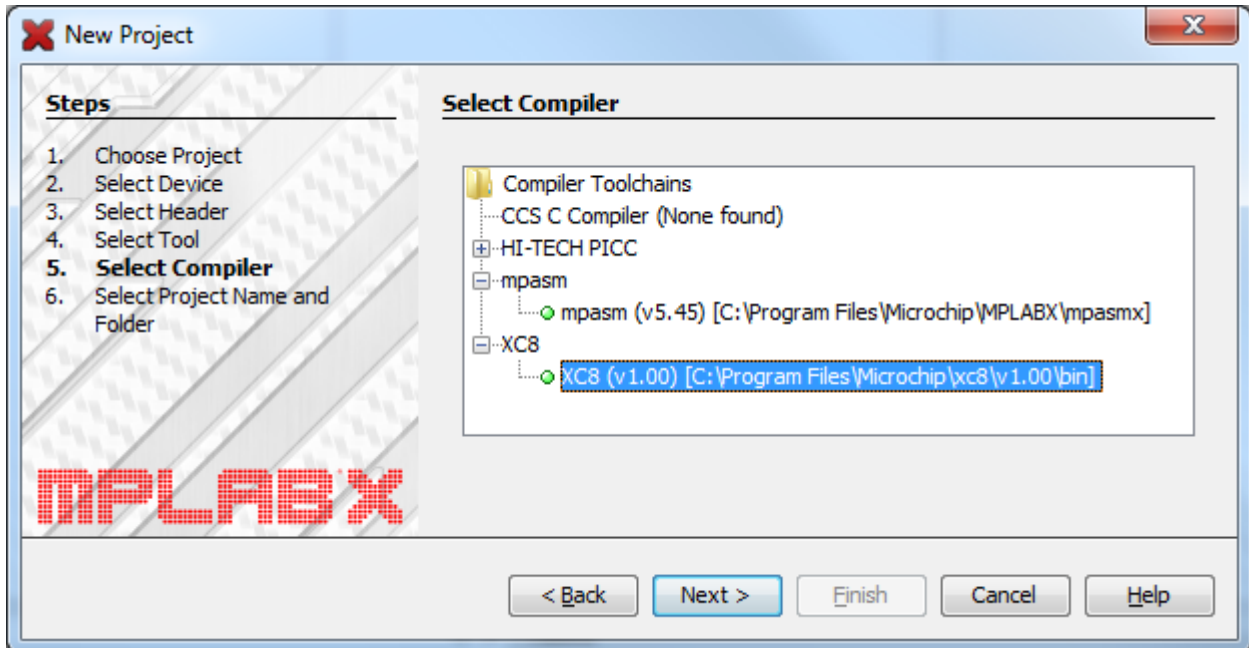
panel shows which compiler will be used for your device; since not every compiler supports every PIC, the toolsuite selects the first driver in the list which supports the device you are compiling for. When you have selected the compiler you wish to use, click “OK” to continue.

You are now ready to start coding!

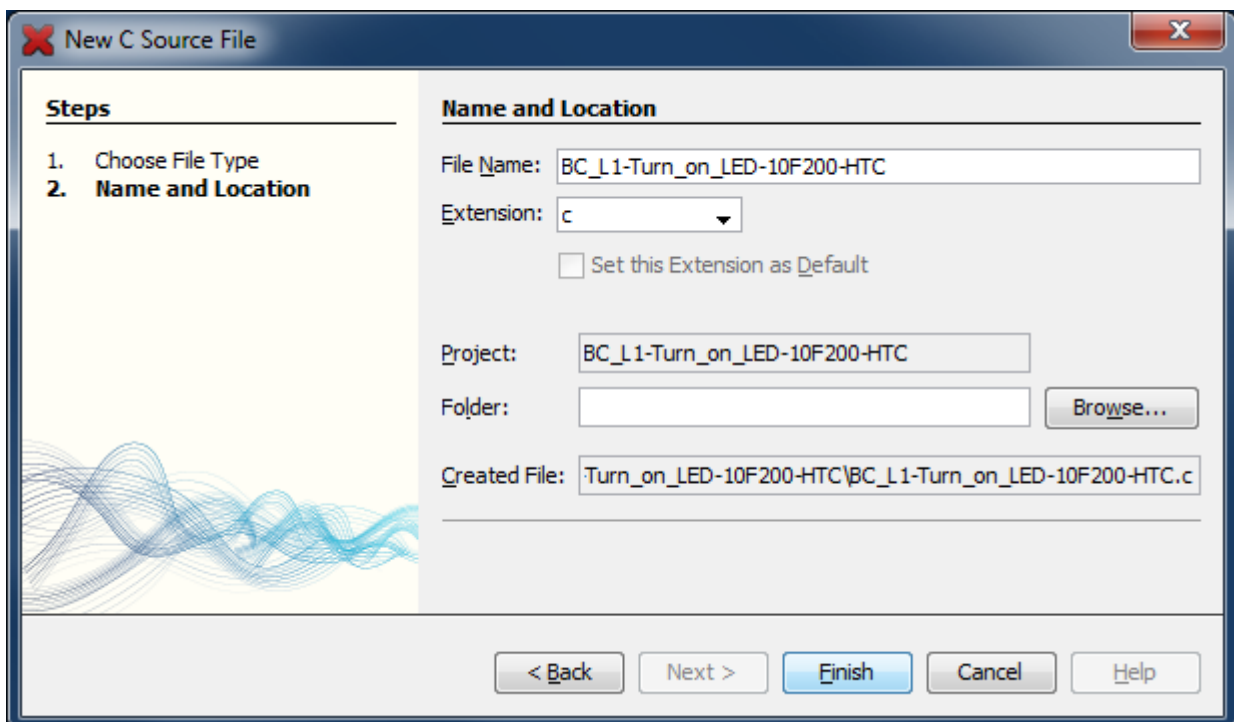
MPLAB X

You should use the New Project wizard to create your new project, as usual.

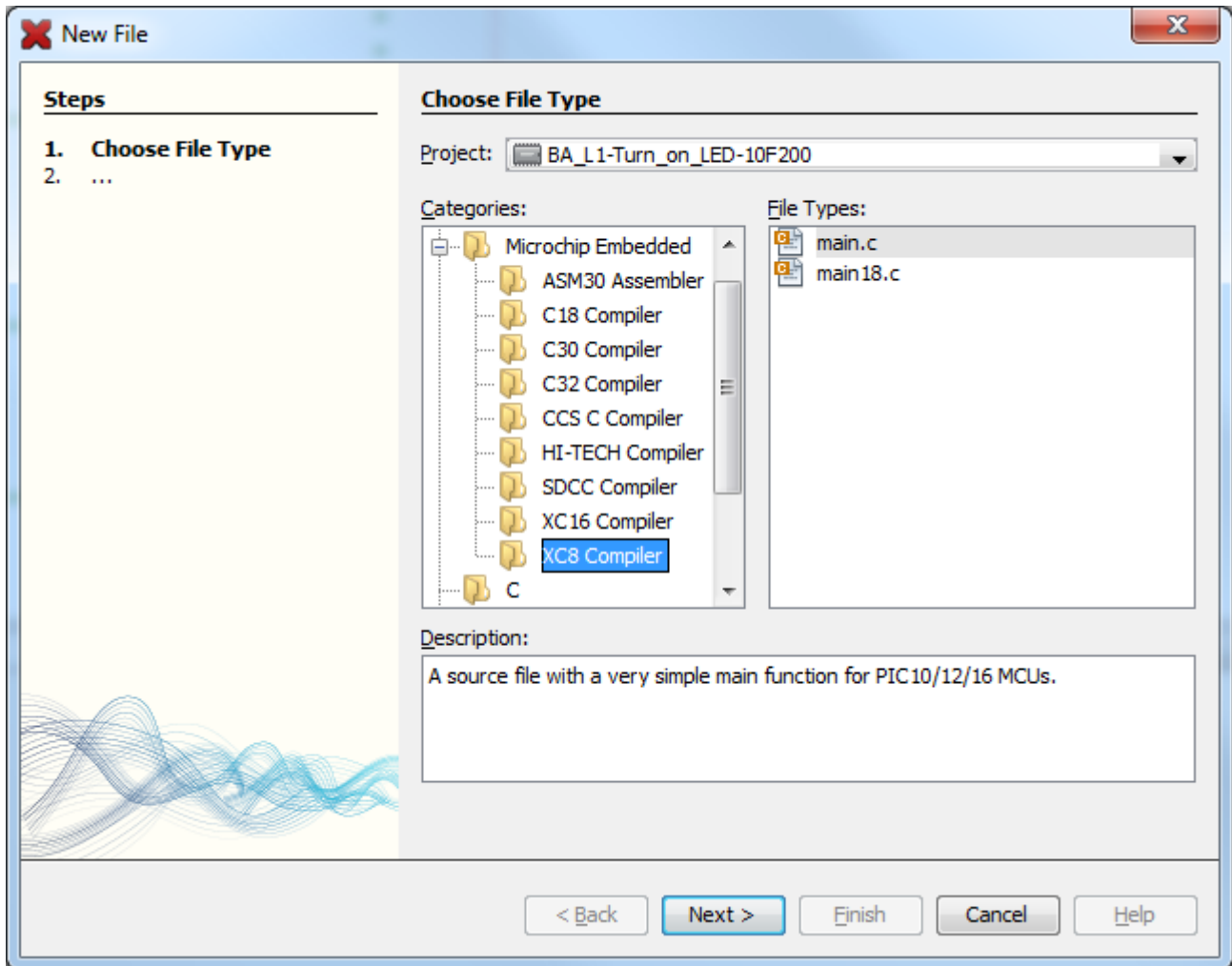
When you reach step 5, “Select Compiler”, select “XC8” (taking care to select the version you wish to use, if you have more than one XC8 compiler installed), to specify that this is an XC8 project:



After completing the wizard, right-click ‘Source Files’ in the project tree within the Projects window, and select “New → C Source File...” to create a ‘.c’ source file in your project folder:

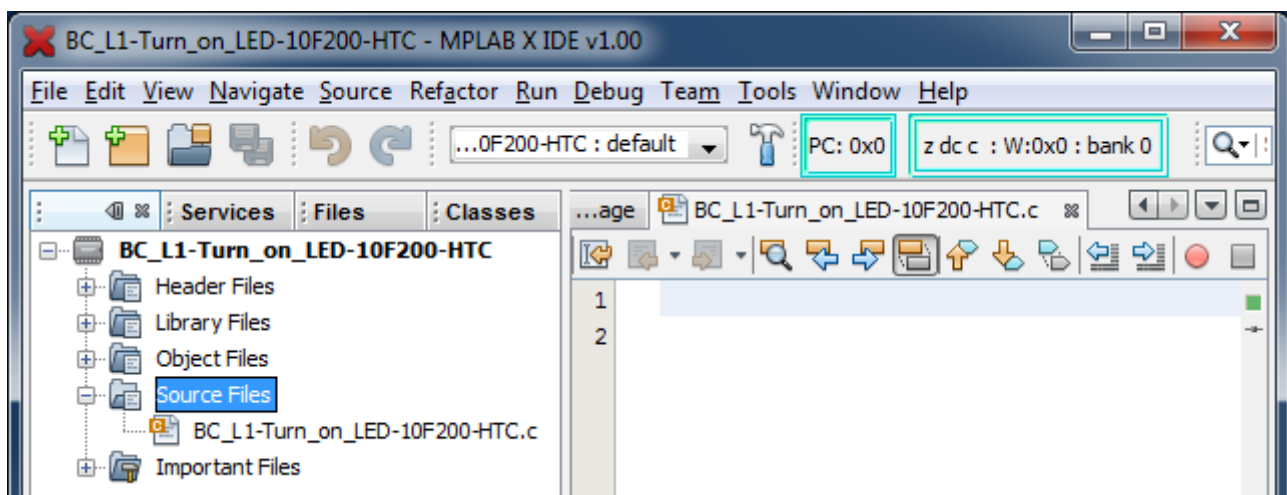


Note that, when you right-click ‘Source Files’ and select the “New” sub-menu, to create a new file and add it to your project, you are presented with a number of options, including any file types you have recently used, and “Other”. This leads you to the “New File” dialog, which allows you to base your new file on an existing template. You’ll find some templates for use with XC8 under ‘Microchip Embedded’:



The “main.c” template is a reasonable start, but it’s slightly different from the C source code style used in these tutorials, so we won’t use it here. Nevertheless, these code templates are a helpful feature of MPLAB X, and as you become more experienced, you can even develop your own.

When you have created a blank C source file, it should appear in the project tree, as usual:



You can now use the editor to start coding!

XC8 Source code

As usual, you should include a comment block at the start of each program or module. Most of the information in the comment block should be much the same, regardless of the programming language used, since it relates to what this application is, who wrote it, dependencies and the assumed environment, such as pin assignments. However, when writing in C, it is a good idea to state which compiler has been used because, as we have seen for data types, C code for microcontrollers is not necessarily easily portable.

So we might use something like:

```

/*****
*
*   Filename:      BC_L1-Turn_on_LED-10F200-HTC.c
*   Date:         7/6/12
*   File Version: 1.1
*
*   Author:       David Meiklejohn
*   Company:      Gooligum Electronics
*
*****
*
*   Architecture: Baseline PIC
*   Processor:    10F200
*   Compiler:     MPLAB XC8 v1.00 (Free mode)
*
*****
*
*   Files required: none
*
*****
*
*   Description:   Lesson 1, example 1
*
*   Turns on LED. LED remains on until power is removed.
*
*****
*
*   Pin assignments:
*       GP1 = indicator LED
*
*****/

```

Note that, as we did our previous assembler code, the processor architecture and device are specified in the comment block. This is important for the XC8 compiler, as there is no way to specify the device in the code; i.e. there is no equivalent to the MPASM ‘list p=’ or ‘processor’ directives. Instead, the processor is specified in the IDE (MPLAB), or as a command-line option.

Most of the symbols relevant to specific processors are defined in header files. But instead of including a specific file, as we would do in assembler, it is normal to include a single “catch-all” file: “xc.h” (or “htc.h”). This file identifies the processor being used, and then calls other header files as appropriate. So our next line, which should be at the start of every XC8 program, is:

```
#include <xc.h>
```

Next, we need to configure the processor.

This can be done with a “*configuration pragma*”.

For the 10F200 version of this example, we would have:

```
// ext reset, no code protect, no watchdog
#pragma config MCLRE = ON, CP = OFF, WDTE = OFF
```

Or, you can use the ‘`__CONFIG`’ macro, in a very similar way to the `__CONFIG` directive in MPASM:

```
__CONFIG(MCLRE_ON & CP_OFF & WDTE_OFF);
```

The symbols are the same in both, but note that the pragma uses ‘=’ (with optional spaces) between each setting, such as ‘MCLRE’, and its value, such as ‘ON’, while the macro uses ‘_’ (with no spaces)⁴.

To see which symbols to use for a given PIC, you need to consult the “pic_chipinfo.html” file, in the “docs” directory within the compiler install directory.

For the 12F508 or 12F509 version⁵, we have:

```
// ext reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & OSC_Intrc);
```

As with most C compilers, the entry point for “user” code is a function called ‘`main()`’.

So an XC8 program will look like:

```
void main()
{
    ; // user code goes here
}
```

Declaring `main()` as `void` isn’t strictly necessary, since any value returned by `main()` is only relevant when the program is being run by an operating system which can act on that return value, but of course there is no operating system here. Similarly it would be more “correct” to declare `main()` as taking no parameters (i.e. `main(void)`), given that there is no operating system to pass any parameters to the program. How you declare `main()` is really a question of personal style.

At the start of our assembler programs, we’ve always loaded the `OSCCAL` register with the factory calibration value (although it is only necessary when using the internal RC oscillator). There is no need to do so when using XC8; the default start-up code, which runs before `main()`, loads `OSCCAL` for us.

XC8 makes the PIC’s special function registers available as variables defined in the header files.

Loading the `TRIS` register with `111101b` (clearing bit 1, configuring `GP1` as an output) is simply:

```
TRIS = 0b111101; // configure GP1 (only) as an output
```

Individual bits, such as `GP1`, can be accessed through bit-fields defined in the header files.

For example, the “pic10f200.h” file header file defines a union called `GPIObits`, containing a structure with bit-field members `GP0`, `GP1`, etc.

⁴ Although the ‘`__CONFIG`’ macro is now (as of XC8 v1.10) considered to be a “legacy” feature, it is still supported and we will continue to use it in these tutorials (the examples were originally written for HI-TECH C).

⁵ The source code for the 12F508 and 12F509 is exactly the same.

So, to set GP1 to '1', we can write:

```
GPIObits.GP1 = 1;          // set GP1 high
```

[Baseline assembler lesson 2](#) explained that setting or clearing a single pin in this way is a “read-modify-write” (“rmw”) operation, which may lead to problems, even though setting GP1 individually, as above, will almost certainly work in this case.

To avoid any potential for rmw problems, we can load the value 000010b into GPIO (setting bit 1, and clearing all the other bits), with:

```
GPIO = 0b000010;         // set GP1 high
```

Finally, we need to loop forever. There are a number of C constructs that could be used for this, but the one used in most of the XC8 sample code (and it's as good as any) is:

```
for (;;)
{
    // loop forever
    ;
}
```

Complete program

Here is the complete 10F200 version of the code to turn on an LED on GP1, for XC8:

```

/*****
*   Description:    Lesson 1, example 1
*
*   Turns on LED.  LED remains on until power is removed.
*
*****/
*
*   Pin assignments:
*       GP1 = indicator LED
*
*****/
#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog
__CONFIG(MCLRE_ON & CP_OFF & WDTE_OFF);

/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    TRIS = 0b111101;          // configure GP1 (only) as an output

    GPIO = 0b000010;         // set GP1 high

    // Main loop
    for (;;)
    {
        // loop forever
        ;
    }
}

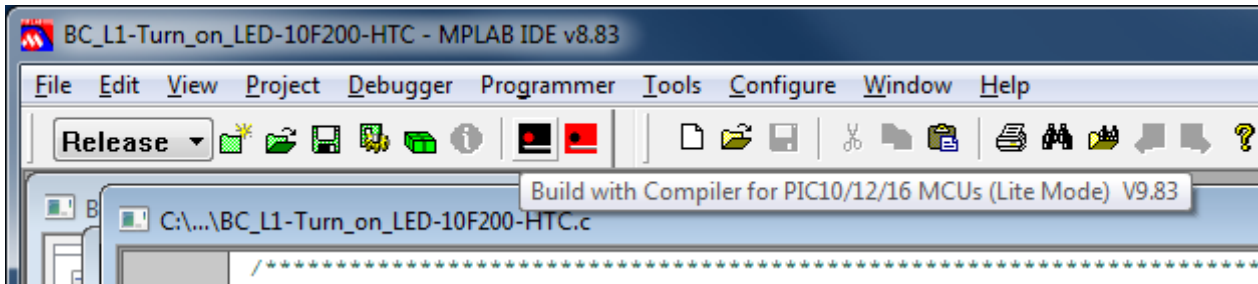
```

The 12F508/509 version is the same, except for the different `__CONFIG` line, given earlier.

Building the project

Whether you use MPLAB 8 or MPLAB X, the process of compiling and linking your code (making or building your project) is essentially the same as for an assembler project (see [baseline assembler lesson 1](#)).

To compile the source code in MPLAB 8, select “Project → Build”, press F10, or click on the “Build” toolbar button:



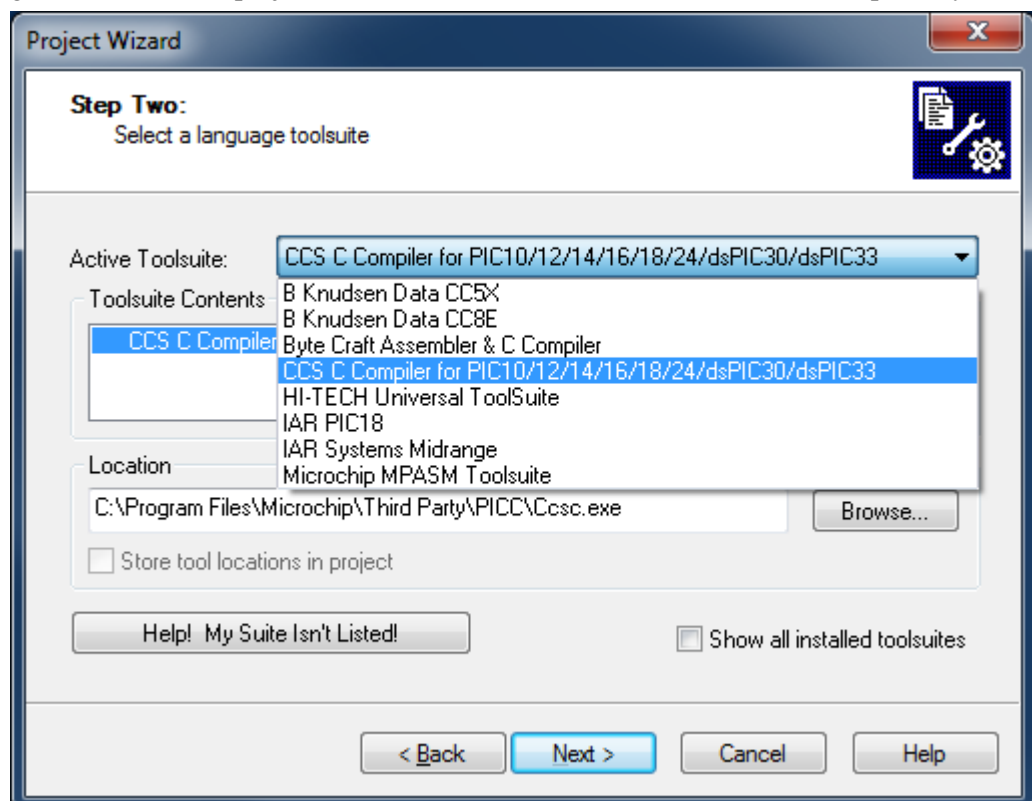
This is equivalent to the assembler “Make” option, compiling all the source files which have changed, and linking the resulting object files and any library functions, creating an output `.hex` file, which can then be programmed into the PIC as normal. The other Project menu item or toolbar button, “Rebuild”, is equivalent to the MPASM “Build All”, recompiling all your source files, regardless of whether they have changed.

Building an XC8 project in MPLAB X is exactly the same as for a MPASM assembler project: click on the “Build” or “Clean and Build” toolbar button, or select the equivalent items in the “Run” menu, to compile and link your code. When it builds without errors and you are ready to program your code into your PIC, select the “Run → Run Main Project” menu item, click on the “Make and Program Device” toolbar button, or simply press F6.

CCS PCB

The process of creating a new CCS PCB project in MPLAB 8 is the same as that for XC8, except that you need to select the “CCS C Compiler” toolsuite, in the project wizard, as shown on the right.

Note that the free version of CCS PCB, bundled with MPLAB 8, cannot be used with MPLAB X. However, if you purchase the most recent version of MPLAB X, you can use MPLAB C to create and build CCS C projects, in the same way as for XC8.



CCS PCB Source code

The comment block at the start of CCS PCB programs can of course be similar to that for any other C compiler (including XC8), but the comments should state that this code is for the CCS compiler.

It's not as important for the comments to state which processor is being used, since, unlike XC8, CCS PCB requires a '#device' directive, used to specify which processor the code is to be compiled for.

Also unlike XC8, there is no "catch-all" header file, so you are expected to include the appropriate '.h' file (found in the "devices" directory within the CCS PCB install directory), which defines all the symbols relevant to the processor you are using. This file will incorporate the appropriate '#device' directive, so you would not normally place that directive separately in your source code. Instead, at the start of every CCS PCB program, you should include a line such as:

```
#include <10F200.h>
```

However, you will find that this file, for the 10F200, defines the pins as PIN_B0, PIN_B1, etc., instead of the more commonly-used GP0, GP1, etc. This is true for the other 10F and 12F PICs as well. So to be able to use the normal symbols, we can add these lines at the start of our code, when working with 10F or 12F PICs:

```
#define GP0 PIN_B0      // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5
```

(if you're using a 10F device, you only need to define GP0 to GP3, because the 10Fs only have four pins)

The '#fuses' directive is used to configure the processor.

For the 10F200, we have:

```
// ext reset, no code protect, no watchdog
#fuses MCLR,NOPROTECT,NOWDT
```

While for the 12F508 or 12F509, we also need to configure the oscillator:

```
// ext reset, no code protect, no watchdog, int RC clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC
```

Again, although this is similar to the __CONFIG directive we know from MPASM, the configuration symbols are different. For example, 'NOPROTECT' instead of '_CP_OFF', and 'INTRC' instead of '_IntRC_OSC'.

To see which symbols to use for a given PIC, you need to consult the header file for that device.

In the same way as XC8, the user program starts with main():

```
void main()
{
    ; // user code goes here
}
```

And, as with XC8, the default start-up code, run before main() is entered, loads the factory calibration value into OSCCAL, so there is no need to write code to do that.

As mentioned earlier, the approach taken by CCS PCB is to make much of the PIC functionality available through built-in functions, reducing the need to access registers directly.

The `output_high()` function loads the TRIS register to configure the specified pin as an output, and then sets that output high. So to make GP1 an output and set it high, we can use simply:

```
output_high(GP1);          // configure GP1 (only) as an output and set high
```

However, as explained for the XC8 version and in detail in [baseline assembler lesson 2](#), setting or clearing a single pin may lead to potential “read-modify-write” problems, and it is safer to write a value to the whole port (GPIO) in a single operation.

CCS C, we can do this with the `output_x()` built-in function, which outputs an entire byte to port x:

```
output_b(0b000010);      // configure GPIO as all output and set GP1 high
```

Note that, as far as CCS is concerned, on a 10F200 or 12F508/509, GPIO is port ‘b’.

Also note that the `output_x()` function configures the port with every pin as an output, before outputting the specified value.

This behaviour of loading TRIS every time an output is made high (or low) makes the code simpler to write, but can be slower and use more memory, so CCS PCB provides a `#use fast_io` directive and `set_tris_x()` functions to override this slower “standard I/O”, but for simplicity, and in the spirit of CCS C programming style, we’ll keep this default behaviour for now.

To loop forever, we could use the `for(;;) {}` loop used in the XC8 example above, but since the standard CCS PCB header files define the symbol ‘TRUE’ (and XC8 doesn’t), we can use:

```
while (TRUE)
{
    // loop forever
}
;
```

Complete program

Here is the complete 10F200 version of the code to turn on an LED on GP1, using CCS PCB:

```

/*****
 *
 *   Description:      Lesson 1, example 1
 *
 *   Turns on LED.   LED remains on until power is removed.
 *
 *****/
 *
 *   Pin assignments:
 *       GP1 = indicator LED
 *
 *****/

#include <10F200.h>

#define GP0 PIN_B0      // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog
#fuses MCLR,NOPROTECT,NOWDT

```

```

/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    output_b(0b000010);    // configure GPIO as all output and set GP1 high

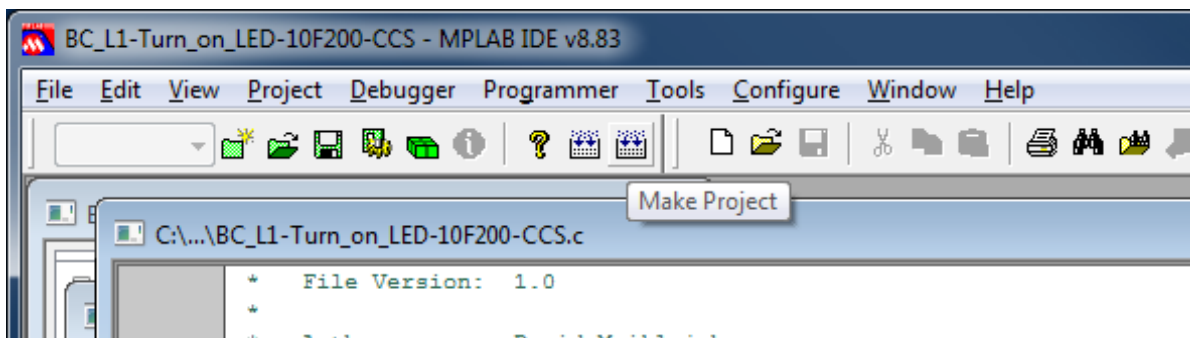
    // Main loop
    while (TRUE)
    {
        // loop forever
        ;
    }
}

```

The 12F508/509 versions are the same, except that you need to `#include` a different header (`12F508.h` or `12F509.h`, instead of `10F200.h`), and change the `#fuses` directive, as shown earlier.

Building the project

To compile the source code in MPLAB 8, select “Project → Build”, press F10, or click on the “Make Project” toolbar button:



This is the CCS equivalent to the XC8 “Build project” option, compiling all changed source files and linking the object files and library functions to create an output ‘.hex’ file, which can be programmed into the PIC as normal.

The other Project menu item or toolbar button, “Build All”, is equivalent to the XC8 “Rebuild” or the MPASM “Build All”, recompiling your entire project, regardless of what’s changed.

Comparisons

Even in an example as simple as turning on a single LED, the difference in approach between XC8 and CCS PCB is apparent.

The XC8 code shows a closer correspondence to the assembler version, with the TRIS register being explicitly written to.

On the other hand, in the CCS PCB example, GPIO is configured as all outputs and written to, through a single built-in function that performs both operations, effectively hiding the existence of the TRIS register from the programmer.

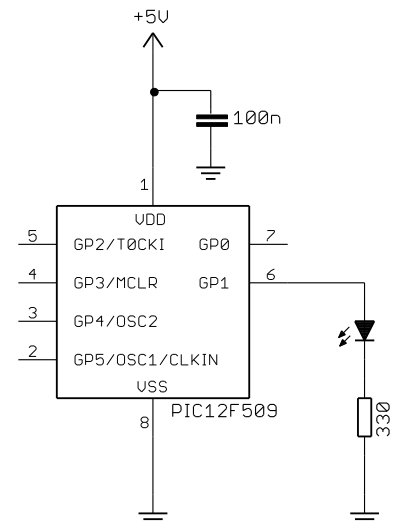
Example 2: Flashing an LED (20% duty cycle)

In [baseline lesson 2](#), we used the same circuits as above, but made the LED flash by toggling the GP1 output. The delay was created by an in-line busy-wait loop.

[Baseline lesson 3](#) showed how to move the delay loop into a subroutine, and to generalise it, so that the delay, as a multiple of 10 ms, is passed as a parameter to the routine, in *W*.

This was demonstrated on the PIC12F509, using the circuit shown on the right. If you are using the Gooligum baseline training board, remember to remove the 10F200 from the '10F' socket, before plugging the 12F509 into the '12F' section of the 14-pin socket.

The example program flashed the LED at 1 Hz with a duty cycle of 20%, by turning it on for 200 ms and then off for 800 ms, and continually repeating.



Here is the main loop from the assembler code from baseline lesson 3:

```
main_loop
    ; turn on LED
    movlw  b'000010'      ; set GP1 (bit 1)
    movwf  GPIO
    ; delay 0.2 s
    movlw  .20           ; delay 20 x 10 ms = 200 ms
    pagesel delay10
    call   delay10
    ; turn off LED
    clrf   GPIO          ; (clearing GPIO clears GP1)
    ; delay 0.8 s
    movlw  .80           ; delay 80 x 10ms = 800ms
    call   delay10

    ; repeat forever
    pagesel main_loop
    goto   main_loop
```

XC8

We've seen how to turn on the LED on GP1, with:

```
GPIObits.GP1 = 1;      // set GP1
```

or

```
GPIO = 0b000010;     // set GP1 (bit 1 of GPIO)
```

And of course, to turn the LED off, it is simply:

```
GPIObits.GP1 = 0;    // clear GP1
```

or

```
GPIO = 0;            // (clearing GPIO clears GP1)
```

These statements can easily be placed within an endless loop, to repeatedly turn the LED on and off. All we need to add is a delay.

XC8 provides a built-in function, '`_delay(n)`', which creates a delay 'n' instruction clock cycles long. The maximum possible delay depends on which PIC you are using, but it is a little over 50,000,000 cycles. With

a 4 MHz processor clock, corresponding to a 1 MHz instruction clock, that's a maximum delay of a little over 50 seconds.

The compiler also provides two macros: ‘`__delay_us()`’ and ‘`__delay_ms()`’, which use the ‘`_delay(n)`’ function create delays specified in μ s and ms respectively. To do so, they reference the symbol “`_XTAL_FREQ`”, which you must define as the processor oscillator frequency, in Hertz.

Since our PIC is running at 4 MHz, we have:

```
#define _XTAL_FREQ 4000000 // oscillator frequency for _delay()
```

Then, to generate a 200 ms delay, we can write:

```
__delay_ms(200); // stay on for 200 ms
```

Complete program

Putting these delay macros into the main loop, we have:

```

/*****
*
* Description: Lesson 1, example 2
*
* Flashes an LED at approx 1 Hz, with 20% duty cycle
* LED continues to flash until power is removed.
*
*****/
*
* Pin assignments:
* GP1 = flashing LED
*
*****/
#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & OSC_InTRC);

#define _XTAL_FREQ 4000000 // oscillator frequency for _delay()

/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    TRIS = 0b111101; // configure GP1 (only) as an output

    // Main loop
    for (;;)
    {
        GPIO = 0b000010; // turn on LED on GP1 (bit 1)
        __delay_ms(200); // stay on for 200 ms
        GPIO = 0; // turn off LED (clearing GPIO clears GP1)
        __delay_ms(800); // stay off for 800 ms
    } // repeat forever
}

```

CCS PCB

In the previous example, we turned on the LED with:

```
output_high(GP1);          // set GP1
or
output_b(0b000010);      // set GP1 (bit 1 of GPIO)
```

Similarly, the LED can be turned off by:

```
output_low(GP1);         // clear GP1
or
output_b(0);            // (clearing GPIO clears GP1)
```

In a similar way to XC8, CCS PCB provides built-in delay functions: 'delay_us()' and 'delay_ms()', which create delays of a specified number of μ s and ms respectively. They accept a 16-bit unsigned value (0-65535) as a parameter. Unlike the XC8 macros, which can only generate a constant delay, the CCS delay functions accept either a variable or a constant as a parameter.

Since the functions are built-in, there is no need to include any header files before using them. But you must still specify the processor clock speed, so that the delays can be created correctly.

This is done using the '#use delay' pre-processor directive to the processor oscillator frequency, in Hertz.

For example, since our PIC is running at 4 MHz, we have:

```
#use delay (clock=4000000)      // oscillator frequency for delay_ms()
```

To create a 200 ms delay, we can then use:

```
delay_ms(200);                // stay on for 200 ms
```

Complete program

Here is the complete code to flash an LED on GP1, with a 20% duty cycle, using CCS PCB:

```

/*****
*
*   Description:    Lesson 1, example 2
*
*   Flashes an LED at approx 1 Hz, with 20% duty cycle
*   LED continues to flash until power is removed.
*
*****/
*
*   Pin assignments:
*   GP1 = flashing LED
*
*****/

#include <12F509.h>

#define GP0 PIN_B0          // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5

```

```

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, int RC clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC

#use delay (clock=4000000)      // oscillator frequency for delay_ms()

/***** MAIN PROGRAM *****/
void main()
{
    // Main loop
    while (TRUE)
    {
        output_b(0b000010);    // turn on LED on GP1 (bit 1)

        delay_ms(200);        // stay on for 200ms

        output_b(0);          // turn off LED (clearing GPIO clears GP1)

        delay_ms(800);        // stay off for 800ms

    }                          // repeat forever
}

```

Example 3: Flashing an LED (50% duty cycle)

The first LED flashing example in [baseline assembler lesson 2](#) used an XOR operation to flip the GP1 bit every 500 ms, creating a 1 Hz flash with a 50% duty cycle.

In that example, we used a *shadow register* to maintain a copy of the port register (GPIO), and flipped the shadow bit corresponding to GP1, instead of working on port directly. As that lesson explained, this was to avoid potential problems due to read-modify-write operations on the port bits. If you're a little hazy on this concept, it would be a good idea to review that section of [baseline assembler lesson 2](#).

As noted earlier, when you use a statement like 'GPIObits.GP1 = 1' in XC8, or 'output_high(GP1)' in CCS PCB, the compilers translate those statements into bit set or clear instructions, acting directly on the port registers, which may lead to read-modify-write problems.

Note: Any C statements which directly modify individual port bits may be subject to read-modify-write considerations.

To avoid such problems, shadow variables can be used in C programs, in the same way that shadow registers are used in assembler programs.

To demonstrate this, we can continue to use the circuit from the previous example, and model our code on the corresponding example from [baseline assembler lesson 3](#):

```

;***** Initialisation
start
    movlw    b'111101'        ; configure GP1 (only) as an output
    tris    GPIO

    clrf    sGPIO            ; start with shadow GPIO zeroed

;***** Main loop

```

```

main_loop
    ; toggle LED on GP1
    movf    sGPIO,w          ; get shadow copy of GPIO
    xorlw   b'000010'       ; toggle bit corresponding to GP1 (bit 1)
    movwf   sGPIO           ; in shadow register
    movwf   GPIO            ; and write to GPIO
    ; delay 0.5 s
    movlw   .50             ; delay 50 x 10 ms = 500 ms
    pagesel delay10         ; -> 1 Hz flashing at 50% duty cycle
    call    delay10

    ; repeat forever
    pagesel main_loop
    goto    main_loop

```

XC8

To toggle GP1, you could use the statement:

```
GPIObits.GP1 = ~GPIObits.GP1;
```

or:

```
GPIObits.GP1 = !GPIObits.GP1;
```

This statement is also supported:

```
GPIObits.GP1 = GPIObits.GP1 ? 0 : 1;
```

It works because single-bit bit-fields, such as GP1, hold either a '0' or '1', representing 'false' or 'true' respectively, and so can be used directly in a conditional expression like this.

However, since these statements modify individual bits in GPIO, to avoid potential read-modify-write issues we'll instead use a shadow variable, which can be declared and initialised with:

```
uint8_t    sGPIO = 0;          // shadow copy of GPIO
```

This makes it clear that the variable is an unsigned, eight-bit integer. We could have declared this as an 'unsigned char', or simply 'char' (because 'char' is unsigned by default), but you can make your code clearer and more portable by using the C99 standard integer types defined in the "stdint.h" header file.

To define these standard integer types, add this line toward the start of your program:

```
#include <stdint.h>
```

This variable declaration could be placed within the main() function, which is what you should do for any variable that is only accessed within main(). However, a variable such as a shadow register may need to be accessed by other functions. For example, it's quite common to place all of your initialisation code into a function called init(), which might initialise the shadow register variables as well as the ports, and your main() code may also need to access them. It is often best to define such variables as global (or "external") variables toward the start of your code, before any functions, so that they can be accessed throughout your program.

But remember that, to make your code more maintainable and to minimise data memory use, you should declare any variable which is only used by one function, as a local variable within that function.

We'll see examples of that later, but in this example we'll define sGPIO as a global variable.

Flipping the shadow copy of GP1 and updating GPIO, can then be done by:

```
sGPIO ^= 0b000010;    // toggle shadow bit corresponding to GP1
GPIO = sGPIO;        // write to GPIO
```

Complete program

Here is how the XC8 code to flash an LED on GP1, with a 50% duty cycle, fits together:

```

/*****
 *
 * Description: Lesson 1, example 3
 *
 * Flashes an LED at approx 1 Hz.
 * LED continues to flash until power is removed.
 *
 *****/
 *
 * Pin assignments:
 * GP1 = flashing LED
 *
 *****/
#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & OSC_InTRC);

#define _XTAL_FREQ 4000000 // oscillator frequency for _delay()

/***** GLOBAL VARIABLES *****/
uint8_t sGPIO = 0; // shadow copy of GPIO

/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    TRIS = 0b111101; // configure GP1 (only) as an output

    // Main loop
    for (;;)
    {
        // toggle LED on GP1
        sGPIO ^= 0b000010; // toggle shadow bit corresponding to GP1
        GPIO = sGPIO; // write to GPIO

        // delay 500 ms
        __delay_ms(500);

    } // repeat forever
}

```


CCS PCB

CCS PCB provides a built-in function specifically for toggling an output pin: `output_toggle()`. To toggle GP1, all that is needed is:

```
output_toggle(GP1);
```

But since this function performs a read-modify-write operation on GPIO, we'll use a shadow variable, which can be declared and initialised with:

```
unsigned int8   sGPIO = 0;           // shadow copy of GPIO
```

CCS PCB doesn't come with an equivalent to `stdint.h`, so we can't use the C99 standard `uint8_t` type, as we did with XC8. We could have declared this variable as a `char`, and that would be ok, but by declaring it as an `unsigned int8`, it's very clear that this variable is an unsigned, eight-bit integer.

And, as in the XC8 example, we'll define this as a global variable, outside `main()`, so that it can be accessed by any other functions you add to the code in future.

Toggling the shadow copy of GP1 is then the same as for XC8:

```
sGPIO ^= 0b000010;           // toggle shadow bit corresponding to GP1
```

To write the result to GPIO, we can use the `output_b()` built-in function, as before:

```
output_b(sGPIO);           // write to GPIO
```

[Recall that CCS PCB refers to GPIO on the 10F and 12F PICs as port B.]

Complete program

Here is the complete CCS PCB code to flash an LED on GP1, with a 50% duty cycle:

```

/*****
*
*   Description:      Lesson 1, example 3
*
*   Flashes an LED at approx 1 Hz.
*   LED continues to flash until power is removed.
*
*****
*
*   Pin assignments:
*   GP1 = flashing LED
*
*****/

#include <12F509.h>

#define GP0 PIN_B0           // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, int RC clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC

```

```
#use delay (clock=4000000)      // oscillator frequency for delay_ms()

/***** GLOBAL VARIABLES *****/
unsigned int8  sGPIO = 0;      // shadow copy of GPIO

/***** MAIN PROGRAM *****/
void main()
{
    // Main loop
    while (TRUE)
    {
        // toggle LED on GP1
        sGPIO ^= 0b000010;      // toggle shadow bit corresponding to GP1
        output_b(sGPIO);        // write to GPIO

        // delay 500ms
        delay_ms(500);

    }    // repeat forever
}
```

Comparisons

Although this is a very small, simple application, it is instructive to compare the source code size (lines of code⁶) and resource utilisation (program and data memory usage) for the two compilers and the assembler version of this example from [baseline lesson 3](#).

Source code length is a rough indication of how difficult or time-consuming a program is to write. We expect that C code is easier and quicker to write than assembly language, but that a C compiler will produce code that is bigger or uses memory less efficiently than hand-crafted assembly. But is this true?

It's also interesting to see whether the delay functions provided by the C compilers generate accurately-timed delays, and how their accuracy compares with our assembler version.

Memory usage is reported correctly by MPLAB for assembler and XC8 projects, but note that the CCS PCB compiler does not accurately report data memory usage to MPLAB. We can get it from the '*.lst' file generated by the CCS compiler.

The MPLAB simulator⁷ can be used to accurately measure the time between LED flashes – ideally it would be exactly 1.000000 seconds, and the difference from that gives us the overall timing error.

Here is the resource usage and accuracy summary for the “Flash an LED at 50% duty cycle” programs:

Flash_LED

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)	Delay accuracy (timing error)
Microchip MPASM	28	34	4	0.15%
XC8 (Free mode)	11	36	4	0.0024%
CCS PCB	9	43	6	0.0076%

⁶ ignoring whitespace, comments, and “unnecessary” lines such as the redefinition of pin names in the CCS C examples

⁷ a topic for a future tutorial?

The assembler version called the delay routine as an external module, so it's quite comparable with the C programs which make use of built-in delay functions. Nevertheless, the assembly language source code is around three times as long as the C versions! This illustrates how much more compact C code can be.

As for C being less efficient – the XC8 version is barely larger than the assembler version, despite having most compiler optimisations disabled in “Free” mode. This is largely because the built-in delay code is highly optimised, but it does show that C is not necessarily inherently inefficient – at least for simple applications like this.

On the other hand, the CCS compiler is noticeably less efficient in this example, generating code 26% bigger than the assembler version.

Finally, note that the time delays in both C versions are amazingly accurate!

Summary

Overall, we have seen that, although XC8 and CCS PCB take quite different approaches, basic digital output operations can be expressed succinctly using either C compiler.

We saw that the CCS approach is to use built-in functions to perform operations which may take a number of statements in XC8 to accomplish (such as configuring pin direction and outputting a value in a single statement). Whether this approach is better is largely a matter of personal style, although having so many built-in functions available can make development much easier.

Whichever compiler you use, it could be argued that, because the C code is significantly shorter than corresponding assembler code, with the program structure more readily apparent, C programs are more easily understood, faster to write, and simpler to debug, than assembler.

So why use assembler? One argument is that, because assembler is closer to the hardware, the developer benefits from having a greater understanding of exactly what the hardware is doing; there are no unexpected or undocumented side effects, no opportunities to be bitten by bugs in built-in or library functions. This argument may apply to CCS PCB, which as we have seen, tends to hide details of the hardware from the programmer; it is not always apparent what the program is always doing “behind the scenes”. But it doesn't really apply to XC8, which exposes all the PIC's registers as variables, and the programmer has to modify the register contents in the same way as would be done in assembler.

Although it's not really apparent in the comparison table above, C compilers consistently use more resources than assembler (for equivalent programs). There comes a point, as programs grow, that a C program will not fit into a particular PIC, while the same program would fit if it had been written in assembler. In that case, the choice is to write in assembler, or use a more expensive PIC. For a one-off project, a more expensive chip probably makes sense, whereas for volume production, using resources efficiently by writing in assembly is the right choice. And if you need to write a really tight loop, where every instruction cycle counts, assembly may be the only viable choice. Although again, using a faster, but more expensive chip may be a better solution, unless your application is high-volume.

If this is a hobby for you, then it's purely a question of personal preference, because as we have seen, both the Microchip and CCS “free” C compilers, as well as assembler, are viable options.

In addition to providing an output (such as a blinking LED), PIC applications usually have to respond sensors and/or user input.

In the [next lesson](#) we'll see how to use our C compilers to read and respond to switches, such as pushbuttons.

And since real switches “bounce”, which can be a problem for microcontroller applications, we'll look at ways to “debounce” them, in software.

Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 2: Reading Switches

The [previous lesson](#) introduced simple digital output, by flashing an LED. That's more useful than it may seem, because, with appropriate circuit changes, the same principles can be readily adapted to turning on and off almost any electrical device.

But most systems also need to respond to user commands or sensor inputs. The simplest form of input is an on/off switch – an example of a digital input: anything that makes or breaks a single connection, or is “on” or “off”, “high” or “low”.

This lesson revisits the material from [baseline assembler lesson 4](#) (which, if you are not familiar with, you should review before you start), showing how to read and respond to a simple pushbutton switch, and handle the inevitable “bouncing” of mechanical switch contacts.

The examples are re-implemented using Microchip's XC8 compiler (running in “Free mode”) and CCS PCB¹, introduced in [lesson 1](#).

This lesson covers:

- Reading digital inputs
- Using internal pull-ups
- Switch debouncing (using a counting algorithm)

with examples for both compilers.

This tutorial assumes a working knowledge of the C language; it does **not** attempt to teach C.

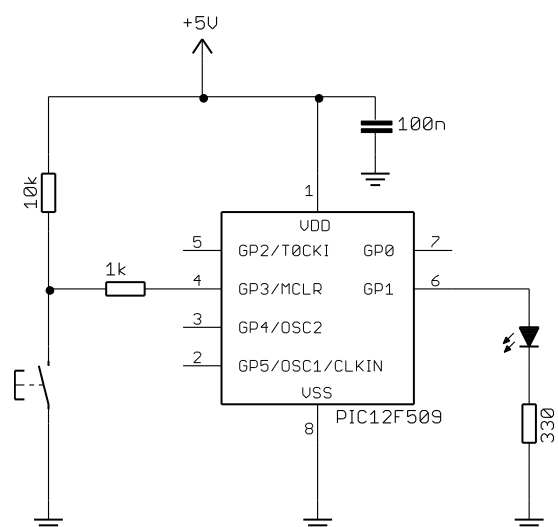
Example 1: Reading Digital Inputs

[Baseline assembler lesson 4](#) introduced digital inputs, using a pushbutton switch in the simple circuit shown on the right.

If you're using the [Gooligum baseline training board](#), you should connect jumper JP3, to bring the 10 kΩ resistor into the circuit, and JP12 to enable the LED on GP1.

The Microchip Low Pin Count Demo Board has a pushbutton, 10 kΩ pull-up resistor and 1 kΩ isolation resistor connected to GP3, as shown. But if you are using that board, you will need to connect an LED to GP1, as described in [baseline assembler lesson 1](#).

The 10 kΩ resistor normally holds the GP3 input high,



¹ XC8 is available as a free download from www.microchip.com, and CCS PCB is bundled for free with MPLAB 8

until the pushbutton is pressed, pulling the input low.

Note: if you are using a PICkit 2 programmer, you must enable ‘3-State on “Release from Reset”’, as described in [baseline assembler lesson 4](#), to allow the pushbutton to pull GP3 low when pressed.

As an initial example, the pushbutton input was copied to the LED output, so that the LED was on, whenever the pushbutton is pressed.

In pseudo-code, the operation is:

```
do forever
    if button down
        turn on LED
    else
        turn off LED
end
```

The assembly code we used to implement this, using a shadow register, was:

```
start
    movlw    b'111101'        ; configure GP1 (only) as an output
    tris     GPIO             ; (GP3 is an input)

loop
    clrf     sGPIO            ; assume button up -> LED off
    btfss   GPIO,3           ; if button pressed (GP3 low)
    bsf     sGPIO,1          ; turn on LED

    movf     sGPIO,w          ; copy shadow to GPIO
    movwf   GPIO

    goto    loop             ; repeat forever
```

XC8

To copy a value from one bit to another, e.g. GP3 to GP1, using XC8, can be done as simply as:

```
GPIObits.GP1 = GPIObits.GP3;           // copy GP3 to GP1
```

But that won't do quite what we want; given that GP3 goes low when the button is pressed, simply copying GP3 to GP1 would lead to the LED being on when the button is up, and on when it is pressed – the opposite of the required behaviour.

We can address that by inverting the logic:

```
GPIObits.GP1 = !GPIObits.GP3;         // copy !GP3 to GP1
or
GPIObits.GP1 = GPIObits.GP3 ? 0 : 1;   // copy !GP3 to GP1
```

This works well in practice, but to allow a valid comparison with the assembly source above, which uses a shadow register, we should not use statements which modify individual bits in GPIO. Instead we should write an entire byte to GPIO at once.

For example, we could write:

```
if (GPIObits.GP3 == 0) // if button pressed
    GPIO = 0b000010;   // turn on LED
else
    GPIO = 0;          // else turn off LED
```

However, this can be written much more concisely using C's conditional expression:

```
GPIO = GPIObits.GP3 ? 0 : 0b000010; // if GP3 high, clear GP1, else set GP1
```

It may seem a little obscure, but this is exactly the type of situation the conditional expression is intended for.

Complete program

Here is the complete XC8 code to turn on an LED when a pushbutton is pressed:

```

/*****
*   Description:      Lesson 2, example 1
*
*   Demonstrates reading a switch
*
*   Turns on LED when pushbutton is pressed
*
*****/
*   Pin assignments:
*       GP1 = indicator LED
*       GP3 = pushbutton switch (active low)
*
*****/

#include <xc.h>

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & WDT_OFF & OSC_IntrRC);

/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    TRIS = 0b111101;           // configure GP1 (only) as an output

    // Main loop
    for (;;)
    {
        // turn on LED only if button pressed
        GPIO = GPIObits.GP3 ? 0 : 0b000010;    // if GP3 high, clear GP1
                                                //else set GP1
    }
}

```

Note that the processor configuration has been changed to disable the external $\overline{\text{MCLR}}$ reset, to allow us to use GP3 as an input.

CCS PCB

Reading a digital input pin with CCS PCB is done through the 'input()' built-in function, which returns the state of the specified pin as a '0' or '1'.

To output a single bit, we could use the 'output_bit()' function. For example:

```
output_bit(GP1, ~input(GP3));
```

This would set GP1 to the inverse of the value on GP3, which is exactly what we want.

But once again, statements like this, which change only one bit in a port, are potentially subject to read-modify-write issues. We should instead use code which writes an entire byte to GPIO (or, as CCS would have it, port B) at once:

```
output_b(input(GP3) ? 0 : 0b000010);    // if GP3 high, clear GP1
                                           // else set GP1
```

Again, using the '?' conditional expression makes this seem a little obscure, but this is very concise and, when you are familiar with these expressions, clear.

Complete program

Here is the complete CCS PCB code to turn on an LED when a pushbutton is pressed:

```

/*****
*
*   Description:    Lesson 2, example 1
*
*   Demonstrates reading a switch
*
*   Turns on LED when pushbutton is pressed
*
*****/
*
*   Pin assignments:
*       GP1 = indicator LED
*       GP3 = pushbutton switch (active low)
*
*****/

#include <12F509.h>

#define GP0 PIN_B0           // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC

/***** MAIN PROGRAM *****/
void main()
{
    // Main loop
    while (TRUE)
    {
        // turn on LED only if button pressed
        output_b(input(GP3) ? 0 : 0b000010);    // if GP3 high, clear GP1
                                                    // else set GP1
    } // repeat forever
}

```

Note again that the processor configuration has been changed to disable the external $\overline{\text{MCLR}}$ reset, so that GP3 is available as an input.

Comparisons

Here is the resource usage summary for the “Turn on LED when pushbutton pressed” programs:

PB_LED

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	18	13	1
XC8 (Free mode)	6	29	2
CCS PCB	5	22	4

At only 5 or 6 lines, the C source code is amazingly succinct – thanks mainly to the use of C’s conditional expression (“?:”).

Example 2: Switch Debouncing

[Baseline lesson 4](#) included a discussion of the switch contact bounce problem, and various hardware and software approaches to addressing it.

The problem was illustrated by an example application, using the circuit from example 1 (above), where the LED is toggled each time the pushbutton is pressed. If the switch is not debounced, the LED toggles on every contact bounce, making it difficult to control.

The most sophisticated software debounce method presented in that lesson was a counting algorithm, where the switch is read (*sampled*) periodically (e.g. every 1 ms) and is only considered to have definitely changed state if it has been in the new state for some number of successive samples (e.g. 10), by which time it is considered to have settled.

The algorithm was expressed in pseudo-code as:

```
count = 0
while count < max_samples
    delay sample_time
    if input = required_state
        count = count + 1
    else
        count = 0
end
```

It was implemented in assembler as follows:

```

; wait for button press, debounce by counting:
db_dn  movlw    .13                ; max count = 10ms/768us = 13
        movwf   db_cnt
        clrf    dcl
dn_dly incfsz   dcl,f              ; delay 256x3 = 768 us.
        goto   dn_dly
        btfsc  GPIO,3            ; if button up (GP3 high),
        goto   db_dn             ; restart count
        decfsz db_cnt,f          ; else repeat until max count reached
        goto   dn_dly
```

This code waits for the button to be pressed (GP3 being pulled low), by sampling GP3 every 768 μ s and waiting until it has been low for 13 times in succession – approximately 10 ms in total.

XC8

To implement the counting debounce algorithm (above) using XC8, the pseudo-code can be translated almost directly into C:

```

db_cnt = 0;
while (db_cnt < 10)
{
    __delay_ms(1);
    if (GPIObits.GP3 == 0)
        db_cnt++;
    else
        db_cnt = 0;
}

```

where the debounce counter variable has been declared as:

```

uint8_t    db_cnt;                // debounce counter

```

Note that, because this variable is only used locally (other functions would never need to access it), it should be declared within `main()`.

Whether you modify this code to make it shorter is largely a question of personal style. Compressed C code, using a lot of “clever tricks” can be difficult to follow.

But note that the `while` loop above is equivalent to the following `for` loop:

```

for (db_cnt = 0; db_cnt < 10;)
{
    __delay_ms(1);
    if (GPIObits.GP3 == 0)
        db_cnt++;
    else
        db_cnt = 0;
}

```

That suggests restructuring the code into a traditional `for` loop, as follows:

```

for (db_cnt = 0; db_cnt <= 10; db_cnt++)
{
    __delay_ms(1);
    if (GPIObits.GP3 == 1)
        db_cnt = 0;
}

```

In this case, the debounce counter is incremented every time around the loop, regardless of whether it has been reset to zero within the loop body. For that reason, the end of loop test has to be changed from ‘<’ to ‘<=’, so that the number of iterations remains the same.

Alternatively, the loop could be written as:

```

for (db_cnt = 0; db_cnt < 10;)
{
    __delay_ms(1);
    db_cnt = (GPIObits.GP3 == 0) ? db_cnt+1 : 0;
}

```

However the previous version seems easier to understand.

Complete program

Here is the complete XC8 code to toggle an LED when a pushbutton is pressed, including the debounce routines for button-up and button-down:

```

/*****
*
*   Description:      Lesson 2, example 2
*
*   Demonstrates use of counting algorithm for debouncing
*
*   Toggles LED when pushbutton is pressed then released,
*   using a counting algorithm to debounce switch
*
*****/
*
*   Pin assignments:
*       GP1 = indicator LED
*       GP3 = pushbutton switch
*
*****/

#include <xc.h>
#include <stdint.h>

#define _XTAL_FREQ 4000000    // oscillator frequency for _delay()

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & WDT_OFF & OSC_IntRC);

/***** GLOBAL VARIABLES *****/
uint8_t      sGPIO;          // shadow copy of GPIO

/***** MAIN PROGRAM *****/
void main()
{
    uint8_t    db_cnt;        // debounce counter

    // Initialisation
    GPIO = 0;                // start with LED off
    sGPIO = 0;               // update shadow
    TRIS = 0b111101;        // configure GP1 (only) as an output

    // Main loop
    for (;;)
    {
        // wait for button press, debounce by counting:
        for (db_cnt = 0; db_cnt <= 10; db_cnt++)
        {
            _delay_ms(1);    // sample every 1 ms
            if (GPIObits.GP3 == 1) // if button up (GP3 high)
                db_cnt = 0;    // restart count
        }
        // until button down for 10 successive reads

        // toggle LED on GP1
        sGPIO ^= 0b000010;    // toggle shadow GP1
        GPIO = sGPIO;        // write to GPIO
    }
}

```

```

// wait for button release, debounce by counting:
for (db_cnt = 0; db_cnt <= 10; db_cnt++)
{
    __delay_ms(1);           // sample every 1 ms
    if (GPIObits.GP3 == 0)  // if button down (GP3 low)
        db_cnt = 0;        // restart count
}
// until button up for 10 successive reads
}
}

```

CCS PCB

To adapt the debounce routine to CCS PCB, the only change needed is to use the `input()` function to read GP3, and to use the `delay_ms()` delay function:

```

for (db_cnt = 0; db_cnt <= 10; db_cnt++)
{
    delay_ms(1);
    if (input(GP3) == 1)
        db_cnt = 0;
}

```

where the debounce counter variable has been declared as:

```

unsigned int8  db_cnt;           // debounce counter

```

Once again, because this variable is only used locally, it should be declared within `main()`.

Complete program

This debounce routine fits into the “toggle an LED when a pushbutton is pressed” program, as follows:

```

/*****
*
* Description:      Lesson 2, example 2
*
* Demonstrates use of counting algorithm for debouncing
*
* Toggles LED when pushbutton is pressed then released,
* using a counting algorithm to debounce switch
*
*****/
*
* Pin assignments:
*   GP1 = indicator LED
*   GP3 = pushbutton switch
*
*****/

#include <12F509.h>

#define GP0 PIN_B0           // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5

/***** CONFIGURATION *****/

```

```

// int reset, no code protect, no watchdog, int RC clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC

#use delay (clock=4000000)      // oscillator frequency for delay_ms()

/***** GLOBAL VARIABLES *****/
unsigned int8   sGPIO = 0;      // shadow copy of GPIO

/***** MAIN PROGRAM *****/
void main()
{
    unsigned int8   db_cnt;      // debounce counter

    // Initialisation
    output_b(0);                // start with LED off
    sGPIO = 0;                  // update shadow

    // Main loop
    while (TRUE)
    {
        // wait for button press, debounce by counting:
        for (db_cnt = 0; db_cnt <= 10; db_cnt++)
        {
            delay_ms(1);        // sample every 1 ms
            if (input(GP3) == 1) // if button up (GP3 high)
                db_cnt = 0;     // restart count
        }                       // until button down for 10 successive reads

        // toggle LED on GP1
        sGPIO ^= 0b000010;      // toggle shadow GP1
        output_b(sGPIO);        // write to GPIO

        // wait for button release, debounce by counting:
        for (db_cnt = 0; db_cnt <= 10; db_cnt++)
        {
            delay_ms(1);        // sample every 1 ms
            if (input(GP3) == 0) // if button down (GP3 low)
                db_cnt = 0;     // restart count
        }                       // until button up for 10 successive reads

    } // repeat forever
}

```

As before, the processor configuration in both the XC8 and CCS programs has been changed to disable the external $\overline{\text{MCLR}}$ reset, so that GP3 is available as an input.

Example 3: Internal (Weak) Pull-ups

As we saw in [baseline assembler lesson 4](#), many PICs include internal “weak pull-ups”, which can be used to pull floating inputs (such as an open switch) high.

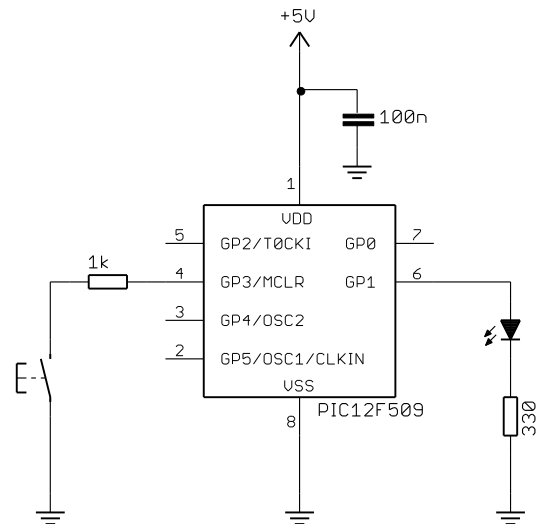
They perform the same function as external pull-up resistors, pulling an input high when a connected switch is open, but supplying only a small current; not enough to present a problem when a closed switch grounds the input.

This means that, on pins where weak pull-ups are available, it is possible to directly connect switches between an input pin and ground, as shown on the right.

To build this circuit, you will need to remove the 10 kΩ external pull-up resistor from the circuit we used previously.

If you have the Gooligum baseline training board, you can simply remove jumper JP3 to disconnect the pull-up resistor from the pushbutton on GP3.

If you're using the Microchip Low Pin Count Demo Board, there's no easy way to take the pull-up resistor out of the circuit. One option is to build the circuit using prototyping breadboard, as shown in [baseline assembler lesson 4](#).



In the baseline (12-bit) PICs, such as the 12F509, the weak pull-ups are not individually selectable; they are either all on, or all off.

To enable the weak pull-ups, clear the $\overline{\text{GPPU}}$ bit in the OPTION register.

In the example assembler program from baseline lesson 4, this was done by:

```
movlw    b'10111111'    ; enable internal pull-ups
          ; -0-----    pullups enabled (/GPPU = 0)
option
```

XC8

To load the OPTION register in XC8, simply assign a value to the variable OPTION.

For example:

```
OPTION = 0b10111111;    // enable internal pull-ups
          //-0-----    pullups enabled (/GPPU = 0)
```

Note that this is commented in a similar way to the assembler version, with '-0-----' making it clear we are concerned with the value of bit 6 ($\overline{\text{GPPU}}$), and that clearing it enables pull-ups.

However, if we use the symbols for register bits, defined in the header files, we can write instead:

```
OPTION = ~nGPPU;        // enable weak pull-ups (/GPPU = 0)
```

To enable weak pull-ups in the “toggle an LED” program from the previous example, simply add this “OPTION =” line into the initialisation routine.

The new initialisation code becomes:

```
// Initialisation
OPTION = ~nGPPU;        // enable weak pull-ups (/GPPU = 0)
GPIO = 0;               // start with LED off
sGPIO = 0;              // update shadow
TRIS = 0b111101;       // configure GP1 (only) as an output
```

CCS PCB

Enabling the internal weak pull-ups using CCS PCB is a little obscure, and not well documented.

The CCS compiler provides a built-in function for enabling pull-ups, 'PORT_x_PULLUPS()', but the documentation (in the online help) for this function states that it is only available for 14-bit (midrange) and 16-bit (18F) PICs. For baseline PICs, we are told:

Note: use SETUP_COUNTERS on PCB parts

However, the documentation for the built-in 'SETUP_COUNTERS()' function makes does not mention the weak pull-ups at all.

To figure this out, we need to go digging in the header files. "12F509.h" includes the following lines:

```
// Timer 0 (AKA RTCC) Functions: SETUP_COUNTERS() or SETUP_TIMER_0(),
...
#define RTCC_INTERNAL    0
...
#define RTCC_DIV_1      8
#define RTCC_DIV_2      0
...
// Constants used for SETUP_COUNTERS() are the above
// constants for the 1st param and the following for
// the 2nd param:
...
// Watch Dog Timer Functions: SETUP_WDT() or SETUP_COUNTERS() (see above)
...
#define WDT_18MS        0x8008
...
#define DISABLE_PULLUPS          0x40 // for 508 and 509 only
#define DISABLE_WAKEUP_ON_CHANGE 0x80 // for 508 and 509 only
```

And here, finally, is a clue.

As explained in [baseline assembler lesson 5](#), the OPTION register in the baseline PICs is mainly used for selecting Timer0 options, including prescaler assignment and prescale ratio. And since the prescaler is shared with the watchdog timer (see [baseline assembler lesson 7](#)), some of these OPTION bits are also used to select watchdog timing options.

That is why the Timer0 and watchdog options are both being set by the 'SETUP_COUNTERS()' function, the use of which is being de-emphasised by CCS, in favour of more specialised built-in functions. But as well as Timer0 and watchdog options, the OPTION register on the baseline PICs also controls the weak pull-up and wake-up on change (see [baseline assembler lesson 7](#)) functions.

Therefore, for the baseline PICs, the 'SETUP_COUNTERS()' function also controls the weak pull-up and wake-up on change functions, in addition to setting timer and watchdog options. It's just not documented very well!

It is not possible to simply enable the weak pull-ups. Instead, we must configure Timer0 (something we'll look at in more detail in the [next lesson](#)); the pull-ups are implicitly enabled by default.

For example:

```
setup_counters(RTCC_INTERNAL, RTCC_DIV_1);
```

To setup the timer without enabling the pull-ups, you explicitly disable them by ORing the 'DISABLE_PULLUPS' symbol with the second parameter.

For example:

```
setup_counters(RTCC_INTERNAL, RTCC_DIV_1 | DISABLE_PULLUPS);
```

To enable weak pull-ups in the “toggle an LED” program from the last example, add this ‘SETUP_COUNTERS ()’ line to the initialisation routine.

Our new initialisation code is:

```
// Initialisation
setup_counters(RTCC_INTERNAL,RTCC_DIV_1); // enable weak pull-ups
output_b(0); // start with LED off
sGPIO = 0; // update shadow
```

Comparisons

Here is the resource usage summary for the “toggle an LED using weak pull-ups” programs:

Toggle_LED+WPU

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	43	36	3
XC8 (Free mode)	21	94	3
CCS PCB	20	82	6

The C programs are less than half as long as the assembler versions, but even the CCS compiler, which has optimisations enabled (unlike the XC8 compiler in “Free mode”), generates code more than twice the size of the hand-written assembler version.

Summary

This lesson has shown that basic digital input operations can readily be performed in C, using either the XC8 or CCS compiler, despite their quite different approaches.

However, we also saw, in example 3, that the use of CCS’s built-in functions does not necessarily make the code easier to follow; the operation of a built-in function may not always be clear, or well-documented. Sometimes, the XC8 approach of directly accessing the PIC registers is actually easier to follow.

In the [next lesson](#) we’ll see how to use these C compilers to configure and access Timer0.

Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 3: Using Timer 0

As demonstrated in the [previous lessons](#), C can be a viable choice for programming digital I/O operations on baseline (12-bit) PICs, although, as we saw, programs written in C can consume significantly more memory (a limited resource on these tiny MCUs) than equivalent programs written in assembler.

This lesson revisits the material from [baseline assembler lesson 5](#) (which you should refer to while working through this tutorial) on the Timer0 module: using it to time events, to maintain the timing of a background task, for switch debouncing, and as a counter.

Selected examples are re-implemented using Microchip's XC8 compiler (running in "Free mode") and CCS PCB¹, introduced in [lesson 1](#). We'll also see the C equivalents of some of the assembler features covered in [baseline assembler lesson 6](#), including macros.

In summary, this lesson covers:

- Configuring Timer0 as a timer or counter
- Accessing Timer0
- Using Timer0 for switch debouncing
- Using C macros

with examples for XC8 and CCS PCB.

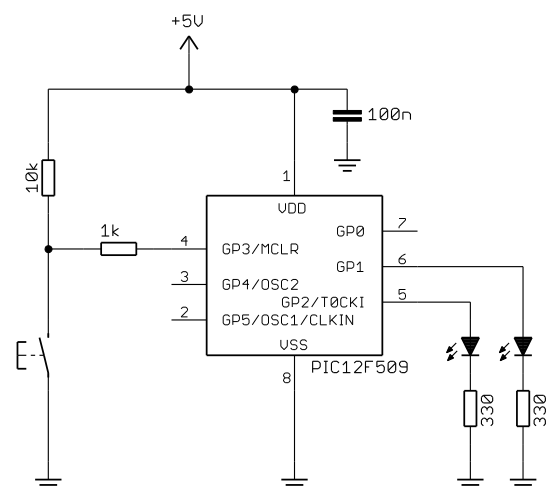
Note that this tutorial series assumes a working knowledge of the C language; it does **not** attempt to teach C.

Example 1: Using Timer0 as an Event Timer

To demonstrate how Timer0 can be used to measure elapsed time, [baseline assembler lesson 5](#) included an example "reaction timer" game, based on the circuit on the right.

To implement this circuit using the [Gooligum baseline training board](#), connect jumpers JP3, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP1 and GP2.

If you are using Microchip's Low Pin Count Demo Board, you will need to connect LEDs to GP1 and GP2, as described in [baseline assembler lesson 1](#).



¹ XC8 is available as a free download from www.microchip.com, and CCS PCB is bundled for free with MPLAB 8

The pushbutton has to be pressed as quickly as possible after the LED on GP2 is lit. If the button is pressed quickly enough (that is, within some predefined reaction time), the LED on GP1 is lit, to indicate ‘success’.

Thus, we need to measure the elapsed time between indicating ‘start’ and detecting a pushbutton press.

An ideal way to do that is to use Timer0, in its timer mode (clocked by the PIC’s instruction clock, which in this example is 1 MHz).

The program flow can be represented in pseudo-code as:

```
do forever
    turn off both LEDs
    delay 2 sec
    indicate start
    clear timer
    wait up to 1 sec for button press
    if button pressed and elapsed time < 200ms
        indicate success
    delay 1 sec
end
```

To use Timer0 to measure the elapsed time, we need to extend its range (normally limited to 65 ms) by adding a counter variable, which is incremented each time the timer overflows (or reaches a certain value). In the example in [baseline lesson 5](#), Timer0 is configured so that it is clocked every 32 μ s, using the 1 MHz instruction clock with a 1:32 prescaler. After 250 counts, 8 ms ($250 \times 32 \mu$ s) will have elapsed; this is used to increment a counter, which is effectively measuring time in 8 ms intervals. This “8 ms counter” can then be checked, when the pushbutton is pressed, to see whether the maximum reaction time has been exceeded.

As explained in that lesson, to select timer mode, with a 1:32 prescaler, we must clear the T0CS and PSA bits, in the OPTION register, and set the PS<2:0> bits to 100. This was done by:

```
movlw    b'11010100'    ; configure Timer0:
; --0-----            timer mode (T0CS = 0)
; ----0---             prescaler assigned to Timer0 (PSA = 0)
; -----100           prescale = 32 (PS = 100)
option   ; -> increment every 32 us
```

The following code was used in [baseline assembler lesson 6](#) to implement the button press / timing test:

```
; wait up to 1 sec for button press
banksel cnt_8ms          ; clear timer (8 ms counter)
clrfsel cnt_8ms          ; repeat for 1 sec:
wait1s  clrfsel TMR0     ; clear Timer0
w_tmr0  ; repeat for 8 ms:
        btfss  BUTTON    ; if button pressed (low)
        goto  wait1s_end ; finish delay loop immediately
        movf   TMR0,w
        xorlw  8000/32    ; (8 ms at 32 us/tick)
        btfss  STATUS,Z
        goto  w_tmr0
        incf   cnt_8ms,f  ; increment 8 ms counter
        movlw  1000/8     ; (1 sec at 8 ms/count)
        xorwf  cnt_8ms,w
        btfss  STATUS,Z
        goto  wait1s
wait1s_end

; check elapsed time
movlw   MAXRT/8          ; if time < max reaction time (8 ms/count)
subwf   cnt_8ms,w
btfss  STATUS,C
bsf     SUCCESS          ; turn on success LED
```

XC8

As we saw in the [previous lesson](#), loading the OPTION register in XC8 is done by assigning a value to the variable OPTION:

```
OPTION = 0b11010100;           // configure Timer0:
    //--0-----             timer mode (T0CS = 0)
    //----0---             prescaler assigned to Timer0 (PSA = 0)
    //-----100           prescale = 32 (PS = 100)
    //                      -> increment every 32 us
```

Note that this has been commented in a way which documents which bits affect each setting, with ‘-’s indicating “don’t care”.

Alternatively, you could express that using the symbols defined in the processor header file (“pic12f509.h” in this case).

For example, since the intent is to clear T0CS and PSA, and to set PS<2:0> to 100, we could make that intent explicit by writing:

```
OPTION = ~T0CS & ~PSA & 0b11111000 | 0b100;
```

(‘0b11111000’ is used to mask off the lower three bits, so that the value of PS<2:0> can be OR’ed in.)

Or, you could write:

```
OPTION = ~T0CS & ~PSA | PS2 & ~PS1 & ~PS0;
```

(specifying the individual PS<2:0> bits)

Which approach you use is largely a question of personal style – and you can adapt your style as appropriate. Although it is often preferable to use symbolic bit names to specify just one or two register bits, using binary constants is quite acceptable if several bits need to be specified at once, especially where some bits need to be set and others cleared (as is the case here) – assuming that it is clearly commented, as above.

The TMR0 register is accessed through a variable, TMR0, so to clear it, we can write:

```
TMR0 = 0;                       // clear timer0
```

and to wait until 8 ms has elapsed:

```
while (TMR0 < 8000/32) // wait for 8 ms (32 us/tick)
    ;
```

The “wait up to one second for button press” routine can then be implemented as:

```
cnt_8ms = 0;
while (BUTTON == 1 && cnt_8ms < 1000/8)
{
    TMR0 = 0;                       // clear timer0
    while (TMR0 < 8000/32) // wait for 8 ms (32 us/tick)
        ;
    ++cnt_8ms;                       // increment 8 ms counter
}
```

where ‘BUTTON’ has been defined as a symbol for ‘GPIObits.GP3’.

As discussed in [baseline assembler lesson 6](#), your code will be easier to understand and maintain if you use symbolic names to refer to pins. If your design changes, you can update the definitions in one place (usually placed at the start of your c, or in a header file). Of course, you may also need to modify your initialisation statements, such as ‘TRIS =’. This is a good reason to keep all your initialisation code in one easily-found place, such as at the start of the program, or in an “init()” function.

Finally, checking elapsed time is simply:

```
if (cnt_8ms < MAXRT/8) // if time < max reaction time (8 ms/count)
    SUCCESS = 1; // turn on success LED
```

Complete program

Here is the complete reaction timer program, using XC8, so that you can see how the various parts fit together:

```

/*****
*
* Description: Lesson 3, example 1
*              Reaction Timer game.
*
* User must attempt to press button within defined reaction time
* after "start" LED lights. Success is indicated by "success" LED.
*
* Starts with both LEDs unlit.
* 2 sec delay before lighting "start"
* Waits up to 1 sec for button press
* (only) on button press, lights "success"
* 1 sec delay before repeating from start
*
*****
*
* Pin assignments:
* GP1 = success LED
* GP2 = start LED
* GP3 = pushbutton switch (active low)
*
*****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & WDT_OFF & OSC_IntrC);

#define _XTAL_FREQ 4000000 // oscillator frequency for _delay()

// Pin assignments
#define START GPIObits.GP2 // LEDs
#define SUCCESS GPIObits.GP1

#define BUTTON GPIObits.GP3 // pushbutton

/***** CONSTANTS *****/
#define MAXRT 200 // Maximum reaction time (in ms)

/***** MAIN PROGRAM *****/
void main()
{
    uint8_t cnt_8ms; // counter: increments every 8 ms

    // Initialisation
    TRIS = 0b111001; // configure GP1 and GP2 (only) as outputs

```

```

OPTION = 0b11010100;           // configure Timer0:
    //--0-----             timer mode (T0CS = 0)
    //----0---              prescaler assigned to Timer0 (PSA = 0)
    //-----100            prescale = 32 (PS = 100)
    //                      -> increment every 32 us

// Main loop
for (;;)
{
    // start with both LEDs off
    GPIO = 0;

    // delay 2 sec
    __delay_ms(2000);           // delay 2000 ms

    // indicate start
    START = 1;                  // turn on start LED

    // wait up to 1 sec for button press
    cnt_8ms = 0;
    while (BUTTON == 1 && cnt_8ms < 1000/8)
    {
        TMR0 = 0;              // clear timer0
        while (TMR0 < 8000/32) // wait for 8 ms (32 us/tick)
            ;
        ++cnt_8ms;             // increment 8 ms counter
    }
    // check elapsed time
    if (cnt_8ms < MAXRT/8)     // if time < max reaction time (8 ms/count)
        SUCCESS = 1;          // turn on success LED

    // delay 1 sec
    __delay_ms(1000);          // delay 1000 ms
} // repeat forever
}

```

CCS PCB

[Lesson 2](#) introduced the ‘`setup_counters()`’ function, which, as we saw, has to be used if you need to enable or disable the weak pull-ups. But its primary purpose (hence the name “setup counters”) is to setup Timer0 and the watchdog timer.

To configure Timer0 for timer mode (using the internal instruction clock), with the prescaler set to 1:32 and assigned to Timer0, you could use:

```
setup_counters(RTCC_INTERNAL, RTCC_DIV_32);
```

However, CCS is de-emphasising the use of ‘`setup_counters()`’, in favour of more specific timer and watchdog setup functions, including ‘`setup_timer_0()`’:

```
setup_timer_0(RTCC_INTERNAL|RTCC_DIV_32);
```

Note that ‘`setup_counters()`’ takes two parameters, while ‘`setup_timer_0()`’ takes a single parameter formed by OR’ing the two symbols.

Both functions work correctly, but ‘`setup_timer_0()`’ produces smaller code, since it is only configuring Timer0, so it is the better choice here.

As we have seen in the previous lessons, the CCS approach is to expose the PIC's functionality through built-in functions, instead of accessing the registers directly.

To set Timer0 to a specific value, use the 'set_timer0()' function, for example:

```
set_timer0(0); // clear timer0
```

To read the current value of Timer0, use the 'get_timer0()' function, for example:

```
while (get_timer0() < 8000/32) // wait for 8ms (32us/tick)
```

The code is then otherwise the same as for XC8.

Complete program

Here is the complete reaction timer program, using CCS PCB:

```

/*****
*
* Description: Lesson 3, example 1
* Reaction Timer game.
*
* User must attempt to press button within defined reaction time
* after "start" LED lights. Success is indicated by "success" LED.
*
* Starts with both LEDs unlit.
* 2 sec delay before lighting "start"
* Waits up to 1 sec for button press
* (only) on button press, lights "success"
* 1 sec delay before repeating from start
*
*****
*
* Pin assignments:
* GP1 = success LED
* GP2 = start LED
* GP3 = pushbutton switch (active low)
*
*****/

#include <12F509.h>

#define GP0 PIN_B0 // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5

/***** CONFIGURATION *****/
// int reset, no code protect, int RC clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC

#use delay (clock=4000000) // oscillator frequency for delay_ms()

// Pin assignments
#define START GP2 // LEDs
#define SUCCESS GP1

#define BUTTON GP3 // pushbutton

```

```

/***** CONSTANTS *****/
#define MAXRT 200 // Maximum reaction time (in ms)

/***** MAIN PROGRAM *****/
void main()
{
    unsigned int8 cnt_8ms; // counter: increments every 8 ms

    // Initialisation
    // configure Timer0: timer mode, prescale = 32 (increment every 32 us)
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_32);

    // Main loop
    while (TRUE)
    {
        // start with both LEDs off
        output_b(0); // clear GPIO

        // delay 2 sec
        delay_ms(2000); // delay 2000 ms

        // indicate start
        output_high(START); // turn on start LED

        // wait up to 1 sec for button press
        cnt_8ms = 0;
        while (input(BUTTON) == 1 && cnt_8ms < 1000/8)
        {
            set_timer0(0); // clear timer0
            while (get_timer0() < 8000/32) // wait for 8 ms (32 us/tick)
                ;
            ++cnt_8ms; // increment 8 ms counter
        }
        // check elapsed time
        if (cnt_8ms < MAXRT/8) // if time < max reaction time (8ms/count)
            output_high(SUCCESS); // turn on success LED

        // delay 1 sec
        delay_ms(1000); // delay 1000 ms
    } // repeat forever
}

```

Comparisons

As we did in the previous lessons, we can compare, for each language/compiler (MPASM assembler, XC8 and CCS PCB), the length of the source code (ignoring comments and white space) versus program and data memory used by the resulting code:

Reaction_timer

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	53	55	4
XC8 (Free mode)	23	83	4
CCS PCB	22	84	6

The C source code is around half as long as the assembler version, but the code generated by both C compilers is significantly larger (around 50%) and uses more data memory – again illustrating the trade-off between programmer efficiency (source code length / complexity) and resource-usage.

Example 2: Background Process Timing

We saw in [baseline assembler lesson 5](#) that one of the key uses of timers is to provide regular timing for “background” processes, while a “foreground” process responds to user signals. Timers are ideal for this, because they continue to run, at a steady rate, regardless of any processing the PIC is doing. On more advanced PICs, a timer is generally used with an interrupt routine, to run these background tasks. But as we’ll see, they can still be useful for maintaining the timing of background tasks, even without interrupts.

The example in baseline lesson 5 used the circuit from example 1, flashing the LED on GP2 at a steady 1 Hz, while lighting the LED on GP1 whenever the pushbutton is pressed.

The 500 ms delay needed for the 1 Hz flash was derived from Timer0 as follows:

- Using a 4 MHz processor clock, providing a 1 MHz instruction clock and a 1 μ s instruction cycle
- Assigning a 1:32 prescaler to the instruction clock, incrementing Timer0 every 32 μ s
- Resetting Timer0 to zero, as soon as it reaches 125 (i.e. every $125 \times 32 \mu$ s = 4 ms)
- Repeating 125 times, creating a delay of $125 \times 4 \text{ ms} = 500 \text{ ms}$.

This was implemented by the following code:

```
main_loop
    ; delay 500 ms
    banksel dly_cnt
    movlw   .125           ; repeat 125 times (125 x 4 ms = 500 ms)
    movwf  dly_cnt
dly500   clrf    TMR0      ; clear timer0
w_tmr0  movf    TMR0,w    ; wait for 4 ms
        xorlw   .125     ; (125 ticks x 32 us/tick = 4 ms)
        btfss  STATUS,Z
        goto   w_tmr0
        decfsz dlycnt,f  ; end 500 ms delay loop
        goto   dly500

        ; toggle flashing LED
        movf   sGPIO,w
        xorlw  b'000100' ; toggle LED on GP2
        movwf  sGPIO    ; using shadow register
        movwf  GPIO

        ; repeat forever
        goto   main_loop
```

And then the code which responds to the pushbutton was placed within the timer wait loop:

```
w_tmr0           ; repeat for 4 ms:
                ; check and respond to button press
        bcf    sGPIO,1 ; assume button up -> indicator LED off
        btfss  GPIO,3  ; if button pressed (GP3 low)
        bsf    sGPIO,1 ; turn on indicator LED
        movf   sGPIO,w ; update port (copy shadow to GPIO)
        movwf  GPIO
        movf   TMR0,w
        xorlw  .125   ; (125 ticks x 32 us/tick = 4 ms)
        btfss  STATUS,Z
        goto   w_tmr0
```

The additional code doesn't affect the timing of the background task (flashing the LED), because there are only a few additional instructions; they are able to be executed within the 32 μ s available between each "tick" of Timer0.

XC8

This assembly code can be implemented in C as:

```
for (;;)
{
    // delay 500 ms while responding to button press
    for (dc = 0; dc < 125; dc++) // repeat 125 times (125 x 4 ms = 500 ms)
    {
        TMR0 = 0; // clear timer0
        while (TMR0 < 125) // repeat for 4 ms (125 x 32 us)
        {
            // check and respond to button press
            sGPIO &= ~(1<<1); // assume button up -> LED off
            if (GP3 == 0) // if button pressed (GP3 low)
                sGPIO |= 1<<1; // turn on LED on GP1
            GPIO = sGPIO; // update port (copy shadow to GPIO)
        }
        // toggle flashing LED
        sGPIO ^= 1<<2; // toggle LED on GP2 using shadow reg
    }
} // repeat forever
```

There is no need to update GPIO after the LED on GP2 is toggled, because GPIO is being continually updated from sGPIO within the inner timer wait loop.

Note the syntax used to set, clear and toggle bits in the shadow GPIO variable, sGPIO:

```
sGPIO |= 1<<1; // turn on LED on GP1
sGPIO &= ~(1<<1); // turn off LED on GP1
sGPIO ^= 1<<2; // toggle LED on GP2
```

We could instead have written:

```
sGPIO |= 0b000010; // turn on LED on GP1
sGPIO &= 0b111101; // turn off LED on GP1
sGPIO ^= 0b000100; // toggle LED on GP2
```

But the left shift ('<<') form more clearly specifies which bit is being operated on.

If we define symbols representing the port bit positions:

```
#define nFLASH 2 // flashing LED on GP2
#define nPRESS 1 // "button pressed" indicator LED on GP1
```

we can write these statements as:

```
sGPIO |= 1<<nPRESS; // turn on indicator LED
sGPIO &= ~(1<<nPRESS); // turn off indicator LED
sGPIO ^= 1<<nFLASH; // toggle flashing LED
```

These symbols can also be used when configuring the port directions:

```
TRIS = ~(1<<nFLASH|1<<nPRESS); // configure LEDs (only) as outputs
```

This makes the code clearer, more general, and therefore more maintainable.

However, this approach doesn't work well on bigger PICs, which have more than one port. You still need to keep track of which port each pin belongs to, and if you change your pin assignments later, you may well need to make a number of changes throughout your code.

A more robust approach is to make use of *bitfields* within C structures.

For example:

```
struct {
    unsigned    GP0      : 1;
    unsigned    GP1      : 1;
    unsigned    GP2      : 1;
    unsigned    GP3      : 1;
    unsigned    GP4      : 1;
    unsigned    GP5      : 1;
} sGPIObits;
```

It is then possible to refer to each bit as a structure member, for example:

```
sGPIObits.GP1 = 1;
```

and if we also defined a symbol such as:

```
#define sPRESS sGPIObits.GP1
```

we can then write this as:

```
sPRESS = 1;
```

That's nice – we have “shadow bits” and we can refer to them easily by symbolic names – but there's still a problem. As well as being able to access individual bits, we also need to be able to refer to the whole shadow register as a single variable, to read or update all the bits at once. After all, that's the whole point of using a shadow register.

We want to be able to change a single bit, as in:

```
sGPIObits.GP1 = 1;    // set shadow GP1
```

and also read the whole shadow register in a single operation, as in:

```
GPIO = sGPIO;        // copy shadow register to port
```

How can we do both?

The C *union* construct is intended for exactly this situation, where we need to access the memory holding a variable in more than one way.

We can define for example:

```
union {
    uint8_t      port;                // shadow copy of GPIO
    struct {
        unsigned    GP0      : 1;
        unsigned    GP1      : 1;
        unsigned    GP2      : 1;
        unsigned    GP3      : 1;
        unsigned    GP4      : 1;
        unsigned    GP5      : 1;
    };
} sGPIO;
```

This allows us to refer to the shadow register as `sGPIO.port`, representing the whole port, in a single operation. For example:

```
sGPIO.port = 0;          // clear shadow register

GPIO = sGPIO.port;     // update port (copy shadow to GPIO)
```

We can also refer to the individual shadow bits as, for example:

```
sGPIO.GP1 = 1;         // set shadow GP1
```

If we define symbols representing these shadow bits:

```
#define sFLASH  sGPIO.GP2      // flashing LED (shadow)
#define sPRESS  sGPIO.GP1      // "button pressed" indicator LED (shadow)
```

we can rewrite the previous bit-manipulation statements as:

```
sPRESS = 1;              // turn on indicator LED
sPRESS = 0;              // turn off indicator LED
sFLASH = !sFLASH;       // toggle flashing LED
```

and, very concisely:

```
sPRESS = !BUTTON;       //      turn on indicator only if button pressed
```

Besides clarity and conciseness, a big advantage of this technique is that, if (on a larger PIC) you were to move one of these functions (such as the flashing LED) to another port, you only need to modify the symbol definition and perhaps your initialisation routine. The rest of your program could stay the same – these statements would still work.

Defining the shadow register as a union incorporating a bitfield structure may seem like a lot of trouble for an apparently small benefit, but it's an elegant approach that will pay off as your applications become more complex.

Complete program

Here is how this shadow register union / bitfield structure definition is used in practice:

```

/*****
*   Description:      Lesson 3, example 2b
*
*   Demonstrates use of Timer0 to maintain timing of background actions
*   while performing other actions in response to changing inputs
*
*   One LED simply flashes at 1 Hz (50% duty cycle).
*   The other LED is only lit when the pushbutton is pressed
*
*   Uses union / bitfield structure to represent shadow register
*
*****/
*
*   Pin assignments:
*   GP1 = "button pressed" indicator LED
*   GP2 = flashing LED
*   GP3 = pushbutton switch (active low)
*
*****/
#include <xc.h>
#include <stdint.h>

```

```

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & WDT_OFF & OSC_IntrC);

// Pin assignments
#define sFLASH  sGPIO.GP2      // flashing LED (shadow)
#define sPRESS  sGPIO.GP1      // "button pressed" indicator LED (shadow)
#define BUTTON  GPIObits.GP3   // pushbutton

/***** GLOBAL VARIABLES *****/
union {
    uint8_t      port;          // shadow copy of GPIO
    struct {
        unsigned  GP0      : 1;
        unsigned  GP1      : 1;
        unsigned  GP2      : 1;
        unsigned  GP3      : 1;
        unsigned  GP4      : 1;
        unsigned  GP5      : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    uint8_t  dc;                // delay counter

    //*** Initialisation

    // configure port
    GPIO = 0;                   // start with all LEDs off
    sGPIO.port = 0;             // update shadow
    TRIS = ~(1<<1|1<<2);      // configure GP1 and GP2 (only) as outputs

    // configure Timer0
    OPTION = 0b11010100;       // configure Timer0:
        //--0-----          timer mode (T0CS = 0)
        //----0---           prescaler assigned to Timer0 (PSA = 0)
        //-----100         prescale = 32 (PS = 100)
        //                  -> increment every 32 us

    //*** Main loop
    for (;;)
    {
        // delay 500 ms while responding to button press
        for (dc = 0; dc < 125; dc++) // repeat 125 times (125 x 4 ms = 500 ms)
        {
            TMR0 = 0;           // clear timer0
            while (TMR0 < 125) // repeat for 4 ms (125 x 32 us)
            {
                sPRESS = !BUTTON; // turn on LED only if button pressed
                GPIO = sGPIO.port; // update port (copy shadow to GPIO)
            }
        }
        // toggle flashing LED
        sFLASH = !sFLASH;      // toggle flashing LED (shadow)
    }
}

```

CCS PCB

There are no new features to introduce. We only need to convert the references to GPIO and TMR0 in the XC8 code into the CCS PCB built-in function equivalents:

```
while (TRUE)
{
    // delay 500 ms while responding to button press
    for (dc = 0; dc < 125; dc++)    // repeat for 500ms (125 x 4ms = 500ms)
    {
        set_timer0(0);            // clear timer0
        while (get_timer0() < 125) // repeat for 4ms (125 x 32us)
        {
            sPRESS = !input(BUTTON); // turn on LED only if button pressed
            output_b(sGPIO.port);    // update port (copy shadow to GPIO)
        }
    }
    // toggle flashing LED
    sFLASH = !sFLASH;              // toggle flashing LED (shadow)
} // repeat forever
```

The shadow register union is defined in much the same way as for XC8:

```
union {
    // shadow copy of GPIO
    unsigned int8    port;
    struct {
        unsigned    GP0    : 1;
        unsigned    GP1    : 1;
        unsigned    GP2    : 1;
        unsigned    GP3    : 1;
        unsigned    GP4    : 1;
        unsigned    GP5    : 1;
    };
} sGPIO;
```

and symbols defined to represent the shadow register bits corresponding to the LEDs and pushbutton:

```
#define sFLASH    sGPIO.GP2    // flashing LED (shadow)
#define sPRESS    sGPIO.GP1    // "button pressed" indicator LED (shadow)
#define BUTTON    PIN_B3       // pushbutton on GP3
```

Note that we have to use ‘PIN_B3’, instead of ‘GP3’. In previous examples, we defined ‘GP3’ as an alias for ‘PIN_B3’, but we can’t do that anymore, because ‘GP3’ has been defined as an element of sGPIO.

Comparisons

Here is the resource usage summary for the “Flash an LED while responding to a pushbutton” programs (the C versions defining the shadow register as a union containing a bitfield structure, as above):

Flash+PB_LED

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	37	31	2
XC8 (Free mode)	28	69	2
CCS PCB	27	47	6

The C source code is comparatively long in this example, because of the shadow register union / bitfield structure definition. It's a big part of the source code – something you wouldn't normally bother with, for such a small program. But we'll keep doing it this way, because it's good practice that will serve us well as our programs become longer, and the extra lines of variable definition won't seem to be such a big deal.

It's interesting to note that the code generated by XC8 (in "Free mode") is now much larger than that generated by the CCS compiler – showing the impact of having most optimisation disabled. The paid-for versions of XC8 could be expected to generate more compact code.

Example 3: Switch debouncing

The [previous lesson](#) demonstrated one method commonly used to debounce switches: sampling the switch state periodically, and only considering it to have definitely changed when it has been in the new state for some minimum number of successive samples.

This "counting algorithm" was expressed as:

```
count = 0
while count < max_samples
    delay sample_time
    if input = required_state
        count = count + 1
    else
        count = 0
end
```

As explained in [baseline assembler lesson 5](#), this can be simplified by using a timer, since it increments automatically:

```
reset timer
while timer < debounce time
    if input ≠ required_state
        reset timer
end
```

This algorithm was implemented in assembler, to wait for and debounce a "button down" event, as follows:

```
wait_dn clrf    TMR0            ; reset timer
chk_dn  btfsc   GPIO,3         ; check for button press (GP3 low)
        goto    wait_dn       ; continue to reset timer until button down
        movf    TMR0,w        ; has 10ms debounce time elapsed?
        xorlw   .157          ; (157 = 10ms/64us)
        btfss  STATUS,Z      ; if not, continue checking button
        goto    chk_dn
```

This code assumes that Timer0 is available, and is in timer mode, with a 1 MHz instruction clock and a 1:64 prescaler, giving 64 μ s per tick.

Of course, since the baseline PICs only have a single timer, it is likely that Timer0 is being used for something else, and so is not available for switch debouncing. But if it is available, it is perfectly reasonable to use it to debounce switches.

This was demonstrated by applying this timer-based debouncing method to the "toggle an LED on pushbutton press" program developed in [baseline assembler lesson 4](#).

XC8

Timer0 can be configured for timer mode, with a 1:64 prescaler, by:

```
OPTION = 0b11010101;           // configure Timer0:
    //--0-----                timer mode (T0CS = 0)
    //----0----                prescaler assigned to Timer0 (PSA = 0)
    //-----101                prescale = 64 (PS = 101)
    //                          -> increment every 64 us
```

This is the same as for the 1:32 prescaler examples, above, except that the **PS<2:0>** bits are set to '101' instead of '100'.

The timer-based debounce algorithm, given above in pseudo-code, is readily translated into C:

```
TMR0 = 0;                       // reset timer
while (TMR0 < 157)              // wait at least 10 ms (157 x 64 us = 10 ms)
    if (GPIObits.GP3 == 1)     // if button up,
        TMR0 = 0;             // restart wait
```

Using C macros

This fragment of code is one that we might want to use a number of times, perhaps modified to debounce switches on inputs other than **GP3**, in this or other programs.

As we saw in [baseline assembler lesson 6](#), the MPASM assembler provides a *macro* facility, which allows a parameterised segment of code to be defined once and then inserted multiple times into the source code.

Macros can also be used when programming in C.

For example, we could define our debounce routine as a macro as follows:

```
#define DEBOUNCE 10*1000/256     // switch debounce count = 10 ms/(256us/tick)

// DbnceLo()
//
// Debounce switch on given input pin
// Waits for switch input to be low continuously for DEBOUNCE*256/1000 ms
//
// Uses: TMR0           Assumes: TMR0 running at 256 us/tick
//
#define DbnceLo(PIN) TMR0 = 0;           /* reset timer           */ \
    while (TMR0 < DEBOUNCE) /* wait until debounce time */ \
        if (PIN == 1) /* if input high, */ \
            TMR0 = 0 /* restart wait */
```

Note that a backslash ('\') is placed at the end of all but the last line, to continue the macro definition over multiple lines. To make the backslashes visible to the C pre-processor, the older `/* */` style comments must be used, instead of the newer `/**` style.

This macro can then be used within your program as, for example:

```
DbnceLo(GPIObits.GP3); // wait until button pressed (GP3 low)
```

You can define macros toward the start of your source code, but as you build your own library of useful macros, you would normally keep them together in one or more header files, such as `"stdmacros.h"`, and reference them from your main program, using the `#include` directive.

Complete program

Here is how this timer-based debounce code (without using macros) fits into the XC8 version of the “toggle an LED on pushbutton press” program:

```

/*****
*   Description:      Lesson 3, example 3a
*
*   Demonstrates use of Timer0 to implement debounce counting algorithm
*
*   Toggles LED when pushbutton is pressed then released
*
*****/
*   Pin assignments:
*       GP1 = flashing LED
*       GP3 = pushbutton switch (active low)
*
*****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & WDT_OFF & OSC_IntrC);

// Pin assignments
#define sFLASH    sGPIO.GP1           // flashing LED (shadow)
#define BUTTON    GPIObits.GP3       // pushbutton

/***** GLOBAL VARIABLES *****/
union {
    uint8_t      port;                // shadow copy of GPIO
    struct {
        unsigned GP0    : 1;
        unsigned GP1    : 1;
        unsigned GP2    : 1;
        unsigned GP3    : 1;
        unsigned GP4    : 1;
        unsigned GP5    : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    GPIO = 0;                // start with LED off
    sGPIO.port = 0;         // update shadow
    TRIS = 0b111101;       // configure GP1 (only) as an output

    // configure timer
    OPTION = 0b11010101;    // configure Timer0:
                            // --0----- timer mode (T0CS = 0)
                            // ----0--- prescaler assigned to Timer0 (PSA = 0)
                            // -----101 prescale = 64 (PS = 101)
                            //          -> increment every 64 us
    
```

```

//*** Main loop
for (;;)
{
    // wait for button press, debounce using timer0:
    TMR0 = 0;                // reset timer
    while (TMR0 < 157)      // wait at least 10 ms (157 x 64 us = 10 ms)
        if (BUTTON == 1)   // if button up,
            TMR0 = 0;      // restart wait

    // toggle LED
    sFLASH = !sFLASH;      // toggle flashing LED (shadow)
    GPIO = sGPIO.port;     // write to GPIO

    // wait for button release, debounce using timer0:
    TMR0 = 0;                // reset timer
    while (TMR0 < 157)      // wait at least 10ms (157 x 64us = 10ms)
        if (BUTTON == 0)   // if button down,
            TMR0 = 0;      // restart wait
}
}

```

CCS PCB

To configure Timer0 for timer mode with a 1:64 prescaler, using CCS PCB, use:

```
setup_timer_0(RTCC_INTERNAL|RTCC_DIV_64);
```

This is the same as we have seen before, except with 'RTCC_DIV_64' instead of 'RTCC_DIV_32'.

The timer-based debounce algorithm can then be expressed as:

```

set_timer0(0);                // reset timer
while (get_timer0() < 157)    // wait at least 10 ms (157 x 64us = 10ms)
    if (input(GP3) == 1)     // if button up,
        set_timer0(0);      // restart wait

```

In the same way as for XC8, this could instead be defined as a macro, as follows:

```

#define DEBOUNCE 10*1000/256    // switch debounce count = 10 ms/(256us/tick)

// DbnceLo()
//
// Debounce switch on given input pin
// Waits for switch input to be low continuously for 10 ms
//
// Uses: TMR0           Assumes: TMR0 running at 256 us/tick
//
#define DbnceLo(PIN)
    set_timer0(0);                /* reset timer */ \
    while (get_timer0() < DEBOUNCE) /* wait until debounce time */ \
        if (input(PIN) == 1)     /* if input high, */ \
            set_timer0(0)        /* restart wait */ \

```

and then called from the main program as, for example:

```
DbnceLo(GP3); // wait until button pressed (GP3 low)
```


Complete program

Here is how the timer-based debounce code (without using macros) fits into the CCS PCB version of the “toggle an LED on pushbutton press” program:

```

/*****
*
* Description: Lesson 3, example 3a
*
* Demonstrates use of Timer0 to implement debounce counting algorithm
*
* Toggles LED when pushbutton is pressed then released
*
*****/
*
* Pin assignments:
* GP1 = flashing LED
* GP3 = pushbutton switch (active low)
*
*****/

#include <12F509.h>

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC

// Pin assignments
#define sFLASH sGPIO.GP1 // flashing LED (shadow)
#define BUTTON PIN_B3 // pushbutton on GP3

/***** GLOBAL VARIABLES *****/
union { // shadow copy of GPIO
    unsigned int8 port;
    struct {
        unsigned GP0 : 1;
        unsigned GP1 : 1;
        unsigned GP2 : 1;
        unsigned GP3 : 1;
        unsigned GP4 : 1;
        unsigned GP5 : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation

    // configure port
    output_b(0); // start with LED off
    sGPIO.port = 0; // update shadow

    // configure Timer0:
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_64); // timer mode, prescale = 64
                                              // -> increment every 64 us

    // Main loop
    while (TRUE)

```

```

{
    // wait for button press, debounce using timer0:
    set_timer0(0);                // reset timer
    while (get_timer0() < 157)    // wait at least 10ms (157x64us = 10ms)
        if (input(BUTTON) == 1) // if button up,
            set_timer0(0);        // restart wait

    // toggle LED
    sFLASH = !sFLASH;            // toggle flashing LED (shadow)
    output_b(sGPIO.port);        // write to GPIO

    // wait until button released (GP3 high), debounce using timer0:
    set_timer0(0);                // reset timer
    while (get_timer0() < 157)    // wait at least 10ms (157x64us = 10ms)
        if (input(BUTTON) == 0) // if button down,
            set_timer0(0);        // restart wait

} // repeat forever
}

```

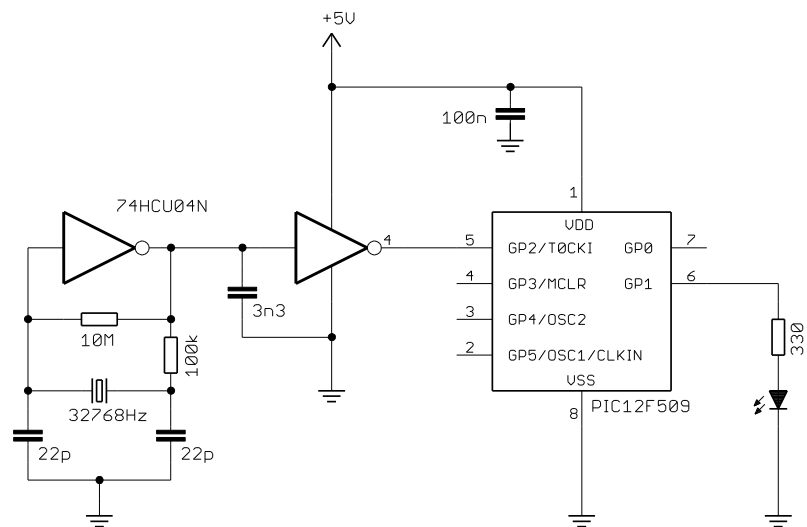
Example 4: Using Counter Mode

So far we've used Timer0 in "timer mode", where it is clocked by the PIC's instruction clock, which runs at one quarter the speed of the processor clock (i.e. 1 MHz when the 4 MHz internal RC oscillator is used). As discussed in [baseline assembler lesson 5](#), the timer can instead be used in "counter mode", where it counts transitions (rising or falling) on the PIC's T0CKI input.

To illustrate how to use Timer0 as a counter, using C, we can use the example from [baseline assembler lesson 5](#). An external 32.768 kHz crystal oscillator (as shown on the right) is used to drive the counter, providing a time base that can be used to flash an LED at a more accurate 1 Hz.

To configure the [Gooligum baseline training board](#) for use with this example, close jumpers JP22 (connecting the 32 kHz clock signal to T0CKI) and JP12 (enabling the LED on GP1).

If you are using Microchip's Low Pin Count Demo Board, you will need to build the oscillator circuit separately, as described in [baseline assembler lesson 5](#).



If the 32.768 kHz clock input is divided (prescaled) by 128, bit 7 of TMR0 will cycle at 1 Hz.

To configure Timer0 for counter mode (external clock on T0CKI) with a 1:128 prescale ratio, we need to set the T0CS bit to '1', PSA to '0' and PS<2:0> to '110'. This was done by:

```

movlw    b'11110110'    ; configure Timer0:
                ; --1-----    counter mode (T0CS = 1)
                ; ----0---    prescaler assigned to Timer0 (PSA = 0)
                ; -----110    prescale = 128 (PS = 110)
option   ; -> increment at 256 Hz with 32.768 kHz input

```

The value of TOSE bit is irrelevant; we don't care if the counter increments on the rising or falling edge of the input clock signal – only the frequency is important. Either edge will do.

Bit 7 of TMR0 (which is cycling at 1 Hz) was then continually copied to GP1, as follows:

```
loop    ; transfer TMR0<7> to GP1
        clrf    sGPIO           ; assume TMR0<7>=0 -> LED off
        btfsc   TMR0,7         ; if TMR0<7>=1
        bsf     sGPIO,1        ; turn on LED

        movf    sGPIO,w         ; copy shadow to GPIO
        movwf   GPIO

        ; repeat forever
        goto    loop
```

XC8

As always, to configure Timer0 using XC8, simply assign the appropriate value to OPTION:

```
OPTION = 0b11110110;           // configure Timer0:
    //--1-----               counter mode (T0CS = 1)
    //----0---                 prescaler assigned to Timer0 (PSA = 0)
    //-----110               prescale = 128 (PS = 110)
    //                          -> increment at 256 Hz with 32.768 kHz input
```

To copy bit 7 of TMR0 to the LED (via a shadow bit), we can use the following construct:

```
sFLASH = 0;                    // assume TMR<7>=0 -> LED off
if (TMR0 & 1<<7)               // if TMR0<7>=1
    sFLASH = 1;                 // turn on LED
```

This works because the expression “1<<7” equals 10000000 binary, so the result of ANDing TMR0 with 1<<7 will only be non-zero if TMR0<7> is set.

Or, we could write this equivalently as:

```
sFLASH = (TMR0 & 1<<7) ? 1 : 0; // sFLASH = 1 only if TMR0<7> = 1
```

or (perhaps more clearly) as :

```
sFLASH = (TMR0 & 1<<7) != 0;    // sFLASH = TMR0<7>
```

Which construct you use is, as ever, a matter of personal style; we'll use the final version here.

Complete program

Here is the XC8 version of the “flash an LED using crystal-driven timer” program:

```
/******
 * Description: Lesson 3, example 4
 *
 * Demonstrates use of Timer0 in counter mode
 *
 * LED flashes at 1 Hz (50% duty cycle),
 * with timing derived from 32.768 kHz input on T0CKI
 *
 * *****
 * Pin assignments:
 * GP1 = flashing LED
 * T0CKI = 32.768 kHz signal
 *
 * *****/
```

```

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, int RC clock
_CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & OSC_IntrC);
// Pin assignments
#define sFLASH  sGPIO.GP1           // flashing LED (shadow)

/***** GLOBAL VARIABLES *****/
union {                               // shadow copy of GPIO
    uint8_t      port;
    struct {
        unsigned GP0    : 1;
        unsigned GP1    : 1;
        unsigned GP2    : 1;
        unsigned GP3    : 1;
        unsigned GP4    : 1;
        unsigned GP5    : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    TRIS = 0b111101;           // configure GP1 (only) as an output

    // configure timer
    OPTION = 0b11110110;       // configure Timer0:
        //--1-----   counter mode (T0CS = 1)
        //----0----   prescaler assigned to Timer0 (PSA = 0)
        //-----110   prescale = 128 (PS = 110)
        //              -> increment at 256 Hz with 32.768 kHz input

    //*** Main loop
    for (;;)
    {
        // TMR0<7> cycles at 1 Hz, so continually copy to LED
        sFLASH = (TMR0 & 1<<7) != 0;    // sFLASH = TMR0<7>

        GPIO = sGPIO.port;           // copy shadow to GPIO
    }
}

```

CCS PCB

To configure Timer0 for counter mode, instead of timer mode, using the CCS PCB `setup_timer_0()` function, use either `'RTCC_EXT_L_TO_H'` (to count low to high input transitions on TOCKI), or `'RTCC_EXT_H_TO_L'`, (for high to low transitions), instead of `'RTCC_INTERNAL'`.

In this example, we don't care if the counter increments on rising or falling edges – it will count at the same rate in either case. So it doesn't matter whether we use `'RTCC_EXT_L_TO_H'` or `'RTCC_EXT_H_TO_L'` here.

We can configure the timer with either:

```
setup_timer_0(RTCC_EXT_L_TO_H|RTCC_DIV_128);
or
setup_timer_0(RTCC_EXT_H_TO_L|RTCC_DIV_128);
```

To copy bit 7 of TMR0 to the LED, we could use the following:

```
sFLASH = 0; // assume TMR<7>=0 -> LED off
if (get_timer0() & 1<<7) // if TMR0<7>=1
    sFLASH = 1; // turn on LED
or
sFLASH = (get_timer0() & 1<<7) != 0; // sFLASH = TMR0<7>
```

However, CCS PCB does provide a facility for accessing (testing or setting/clearing) bits directly.

The bit to be accessed must first be declared as a variable, using the #bit directive.

For example:

```
#bit TMR0_7 = 0x01.7 // bit 7 of TMR0
```

This variable can then be used the same way as any other single-bit variable, and can be assigned directly, making it possible to write simply:

```
sFLASH = TMR0_7;
```

As you can see, defining bit variables in this way makes for straightforward, easy-to-read code. However, CCS discourages this practice; as the help file warns, “*Register locations change between chips*”. For example, the code above assumes that TMR0 is located at address 0x01. If this code is migrated to a PIC with a different address for TMR0, and you forget to change the bit variable definition, the problem may be very hard to find – the compiler wouldn’t produce an error. Your code would simply continue to test bit 7 of whatever register happened to now be at address 0x01.

So although, by using the #bit directive, you can make your code clearer and more efficient, you should use it carefully. Using the CCS built-in functions is safer, and easier to maintain.

Complete program

Here is how the code, using CCS PCB, with the #bit pre-processor directive, for the “flash an LED using crystal-driven timer” fits together:

```
/*
 * Description: Lesson 3, example 4
 *
 * Demonstrates use of Timer0 in counter mode
 *
 * LED flashes at 1 Hz (50% duty cycle),
 * with timing derived from 32.768 kHz input on T0CKI
 *
 *
 * Pin assignments:
 * GP1 = flashing LED
 * T0CKI = 32.768 kHz signal
 */
```

```

#include <12F509.h>

#define TMR0_7 = 0x01.7 // bit 7 of TMR0

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, int RC clock
#define fuses MCLR,NOPROTECT,NOWDT,INTRC

// Pin assignments
#define sFLASH sGPIO.GP1 // flashing LED (shadow)

/***** GLOBAL VARIABLES *****/
union { // shadow copy of GPIO
    unsigned int8 port;
    struct {
        unsigned GP0 : 1;
        unsigned GP1 : 1;
        unsigned GP2 : 1;
        unsigned GP3 : 1;
        unsigned GP4 : 1;
        unsigned GP5 : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation

    // configure Timer0:
    setup_timer_0(RTCC_EXT_L_TO_H|RTCC_DIV_128); // counter mode, prescale = 128
                                                // -> increment at 256 Hz
                                                // with 32.768 kHz input

    // Main loop
    while (TRUE)
    {
        // TMR0<7> cycles at 1 Hz, so continually copy to LED
        sFLASH = TMR0_7;

        output_b(sGPIO.port); // copy shadow to GPIO

    } // repeat forever
}

```

Summary

These examples have demonstrated that Timer0 can be effectively configured and accessed using the XC8 and CCS C compilers, with the program algorithms being able to be expressed quite succinctly in C.

We've also seen that using symbolic names and macros can help make your code more maintainable, and how the union and bitfield structure constructs can be used to make it possible to access both a whole variable and its individual bits, in an elegant way.

In the [next lesson](#) we'll see how these C compilers can be used with processor features such as sleep mode and the watchdog timer, and to select various clock, or oscillator, configurations.

Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 4: Sleep Mode and the Watchdog Timer

Continuing the series on C programming, this lesson revisits material from [baseline lesson 7](#), which examined the baseline PIC architecture's power-saving sleep mode, its ability to wake from sleep when an input changes and the watchdog timer – generally used to automatically restart a crashed program, but also useful for periodically waking the PIC from sleep, for low-power operation. As before, selected examples from that lesson are re-implemented using Microchip's XC8 compiler (running in "Free mode") and CCS PCB¹, introduced in [lesson 1](#).

[Baseline assembler lesson 7](#) also described the various clock, or oscillator, configurations available for the PIC12F508/509 – a topic which does not really need a separate treatment for C, since the programming techniques needed to implement the examples from that lesson have already been covered in lessons [1](#) to [3](#). Only the PIC configuration is different, so this lesson includes a table listing the corresponding configuration word settings between MPASM, XC8 and CCS PCB.

In summary, this lesson covers:

- Sleep mode (power down)
- Wake-up on change (power up on input change)
- The watchdog timer, including periodic wake from sleep
- Configuration word settings

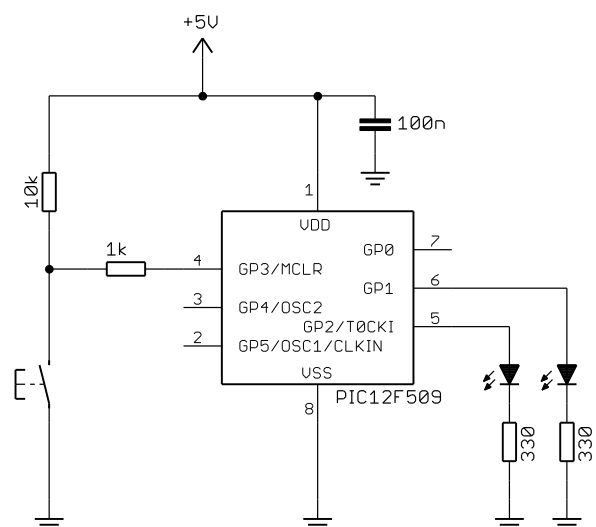
with examples for XC8 and CCS PCB.

Circuit Diagram

The examples in this lesson use the circuit shown on the right, consisting of a PIC12F509 and 100 nF bypass capacitor, with LEDs on GP1 and GP2, and a pushbutton switch on GP3.

If you have the [Gooligum baseline training board](#), connect jumpers JP3, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP1 and GP2. Or, if you are using Microchip's Low Pin Count Demo Board, you will need to connect LEDs to GP1 and GP2, as described in [baseline lesson 1](#).

However, if you want to be able to see how the power consumption is reduced when the PIC is placed into sleep mode, you should use an external power supply,



¹ XC8 is available as a free download from www.microchip.com, and CCS PCB is bundled for free with MPLAB 8

instead of using your PICkit 2 or PICkit 3 to power the circuit. You can then place a multimeter in-line with the power supply, to measure the supply current.

Sleep Mode

As explained in [baseline lesson 7](#), the assembler instruction for placing the PIC into sleep mode is ‘sleep’.

This was demonstrated by the following code, which turns on an LED, waits for a pushbutton press, and then turns off the LED (saving power) before placing the PIC permanently into sleep mode (effectively shutting it down):

```

        ; turn on LED
        bsf      LED

        ; wait for button press
wait_lo btfsc   BUTTON      ; wait until button low
        goto    wait_lo

        ; go into standby (low power) mode
        bcf      LED        ; turn off LED
        sleep      ; enter sleep mode

        goto    $          ; (this instruction should never run)

```

XC8

To place the PIC into sleep mode, XC8 provides a ‘SLEEP()’ macro.

It is defined in the “pic.h” header file (called from the “xc.h” file we’ve included at the start of each XC8 program), as:

```
#define SLEEP() asm("sleep")
```

‘asm()’ is a XC8 statement which embeds a single assembler instruction, in-line, in the C source code. But since ‘SLEEP()’ is provided as a standard macro, it makes sense to use it, instead of the ‘asm()’ statement.

Complete program

The following program shows how the XC8 ‘SLEEP()’ macro is used:

```

/*****
*
* Description: Lesson 4, example 1
*
* Demonstrates sleep mode
*
* Turn on LED, wait for button pressed, turn off LED, then sleep
*
*****/
*
* Pin assignments:
* GP1 = indicator LED
* GP3 = pushbutton (active low)
*
*****/

#include <xc.h>

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & WDT_OFF & OSC_IntRC);

```



```

// Pin assignments
#define LED      GPIObits.GP1    // Indicator LED on GP1
#define nLED     1               // (port bit 1)
#define BUTTON   GPIObits.GP3    // Pushbutton (active low)

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation
    // configure port
    TRIS = ~(1<<nLED);           // configure LED pin (only) as an output

    /*** Main code
    // turn on LED
    LED = 1;

    // wait for button press
    while (BUTTON == 1)         // wait until button low
        ;

    // go into standby (low power) mode
    LED = 0;                    // turn off LED
    SLEEP();                    // enter sleep mode

    for (;;)                    // (this loop should never execute)
        ;
}

```

CCS PCB

Consistent with CCS' stated approach of allowing most tasks to be performed through built-in functions, the PCB compiler provides a function for entering sleep mode: 'sleep()'.

Unlike the XC8 version, this is a built-in function, not a macro. But it's used the same way.

Complete program

Here is the CCS PCB version of the "sleep after pushbutton press" program:

```

*****
*
* Description:    Lesson 4, example 1
*
* Demonstrates sleep mode
*
* Turn on LED, wait for button pressed, turn off LED, then sleep
*
*****
*
* Pin assignments:
*     GP1 = indicator LED
*     GP3 = pushbutton (active low)
*
*****/

#include <12F509.h>

#define GP0 PIN_B0           // define GP pins

```

```

#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC

// Pin assignments
#define LED      GP1           // Indicator LED
#define BUTTON  GP3           // Pushbutton (active low)

/***** MAIN PROGRAM *****/
void main()
{
    // turn on LED
    output_high(LED);

    // wait for button press
    while (input(BUTTON) == 1)    // wait until button low
        ;

    // go into standby (low power) mode
    output_low(LED);             // turn off LED
    sleep();                     // enter sleep mode

    while (TRUE)                 // (this loop should never execute)
        ;
}

```

Wake-up on Change

We saw in [baseline assembler lesson 7](#) that, if the $\overline{\text{GPWU}}$ bit in the OPTION register is cleared, the PIC12F509 will come out of sleep (“wake-up”), if any of the GP0, GP1 or GP3 inputs change.

Note that, in the baseline PIC architecture, wake-up on change can only be enabled on certain pins, and that it is either enabled for all of these pins or for none of them; it is not individually selectable.

This feature can be used in low-power applications, where the PIC spends most of the time sleeping (saving power), waking only to respond to external events which lead to an input change. It’s also useful when designing devices with a “soft” on/off feature, such as the [Gooligum Electronics “Hangman”](#) project, which is based on a baseline PIC (the 16F505).

Note also that it’s important to read any input pins configured for wake on change, and to ensure that they are stable (by debouncing any switches) just prior to entering sleep mode, to avoid the PIC immediately waking up.

Baseline PICs restart when they wake from sleep, recommencing execution at the reset vector (0x000), in the same way they do when first powered on (power-on reset) or following an external reset on $\overline{\text{MCLR}}$.

If the reset was due to a wake on change, it may be necessary to debounce whichever input changed, to avoid the program responding to spurious input transitions from the switch bounce. You could perform this debounce “just in case”, regardless of why the PIC (re)started.

But in some cases you'll want your program to behave differently if it was restarted by a wake on change, can be done by testing the GPWUF flag in the STATUS register. GPWUF is set to '1' only if a wake on change reset has occurred.

These concepts were demonstrated in [baseline lesson 7](#), through an example similar to that in the sleep mode section, above, and using the same circuit. One of the LEDs is turned on and then, when the pushbutton is pressed, it is turned off and the PIC is put into sleep mode. But in this example, wake on change is enabled, so that when the pushbutton is pressed again (changing the input on GP3), the program restarts and the LED is turned on again. If GPWUF is set, a second LED is lit, to indicate that a wake on change happened.

This was implemented in assembler as:

```

; turn on LED
bsf    LED

; test for wake-on-change reset
btfss  STATUS,GPWUF    ; if wake-up on change has occurred,
goto   wait_lo
bsf    WAKE             ; turn on wake-up indicator
DbnceHi BUTTON         ; wait for button to stop bouncing

; wait for button press
wait_lo btfsc  BUTTON    ; wait until button low
goto   wait_lo

; go into standby (low power) mode
clrf   GPIO            ; turn off LEDs

DbnceHi BUTTON         ; wait for stable button release

sleep                                     ; enter sleep mode

```

XC8

To enable wake-up on change using XC8, simply ensure that the $\overline{\text{GPWU}}$ bit in the OPTION register is cleared, for example:

```

OPTION = 0b01000111;    // configure wake-up on change and Timer0:
//0-----             // enable wake-up on change (/GPWU = 0)
//--0-----           // timer mode (T0CS = 0)
//----0---            // prescaler assigned to Timer0 (PSA = 0)
//-----111           // prescale = 256 (PS = 111)
//                    // -> increment every 256 us

```

Testing the GPWUF flag is simple; it is defined as a bit-field in the header files provided with the compiler (as all the special function register bit are), and can be accessed directly:

```

if (STATUSbits.GPWUF)    // if wake on change has occurred,
{
    WAKE = 1;            // turn on wake-up indicator
    DbnceHi (BUTTON);    // wait for button to stop bouncing
}

```

The test could instead be written more explicitly as:

```

if (STATUSbits.GPWUF == 1) { ... }    // if wake on change has occurred...

```

But it's considered quite acceptable, and perfectly clear, to leave out the '== 1' when testing a flag bit.

Complete program

Here is how the above fragments fit into the program:

```

/*****
*   Description:    Lesson 4, example 2
*
*   Demonstrates wake-up on change
*                   plus differentiation from POR reset
*
*   Turn on LED after each reset
*   Turn on WAKE LED only if reset was due to wake on change
*   then wait for button press, turn off LEDs, debounce, then sleep
*
*****/
*
*   Pin assignments:
*   GP1 = on/off indicator LED
*   GP2 = wake-on-change indicator LED
*   GP3 = pushbutton switch (active low)
*
*****/

#include <xc.h>

#include "stdmacros-HTC.h" // DbnceHi() - debounce switch, wait for high
                          // Requires: TMR0 at 256us/tick

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & WDT_OFF & OSC_IntrC);

// Pin assignments
#define LED      GPIObits.GP1    // LED to turn on/off
#define nLED     1                // (port bit 1)
#define WAKE     GPIObits.GP2    // indicates wake on change condition
#define nWAKE    2                // (port bit 2)
#define BUTTON   GPIObits.GP3    // Pushbutton (active low)

/***** MAIN PROGRAM *****/
void main()
{
    //***** Initialisation

    // configure port
    GPIO = 0;                // start with both LEDs off
    TRIS = ~(1<<nLED|1<<nWAKE); // configure LED pins as outputs

    // configure wake-on-change and timer
    OPTION = 0b01000111;    // configure wake-up on change and Timer0:
        //0-----          enable wake-up on change (/GPWU = 0)
        //--0-----        timer mode (T0CS = 0)
        //----0---          prescaler assigned to Timer0 (PSA = 0)
        //-----111        prescale = 256 (PS = 111)
        //                  -> increment every 256 us

    //***** Main code

    // turn on LED
    LED = 1;

```

```

// test for wake-on-change reset
if (STATUSbits.GPWUF) // if wake on change has occurred,
{
    WAKE = 1; // turn on wake-up indicator
    DbncHi(BUTTON); // wait for button to stop bouncing
}

// wait for button press
while (BUTTON == 1) // wait until button low
    ;

// go into standby (low power) mode
GPIO = 0; // turn off both LEDs

DbncHi(BUTTON); // wait for stable button release

SLEEP(); // enter sleep mode
}

```

CCS PCB

Although the CCS PCB compiler provides built-in functions to perform most tasks, *there are no built-in functions for explicitly enabling wake on change, or for detecting a wake on change reset.*

We saw in [lesson 2](#) that weak pull-ups are enabled implicitly whenever Timer0 is configured, and that the only way to disable weak pull-ups is to use the `setup_counters()` function with an additional 'DISABLE_PULLUPS' symbol.

Similarly, wake-up on change is enabled implicitly whenever Timer0 is configured, whether you use `setup_timer_0()` or `setup_counters()`.

To setup the timer without enabling wake-up on change, you must use the `setup_counters()` function with the 'DISABLE_WAKEUP_ON_CHANGE' symbol ORed with the second parameter.

For example:

```
setup_counters(RTCC_INTERNAL, RTCC_DIV_1 | DISABLE_WAKEUP_ON_CHANGE);
```

CCS PCB does not provide any built-in function which can be used to detect that a wake-up on change reset has occurred. Although the compiler does provide a 'restart_cause()' function, which returns a value indicating the cause of the last reset, this does not encompass wake on change resets on the baseline PICs. The "12F509.h" header file does define the symbol 'PIN_CHANGE_FROM_SLEEP', which is presumably intended to be used in detecting a wake-up on pin change, but it is not a valid return code from the 'restart_cause()' function.

So to detect a wake on change reset, we need to use the `#bit` directive, introduced in [lesson 3](#), to allow access to the GPWUF flag, as follows:

```
#bit GPWUF = 0x03.7 // GPWUF flag in STATUS register
```

This flag can then be tested directly, in the same way as we did with XC8:

```

if (GPWUF) // if wake on change has occurred,
{
    output_high(WAKE); // turn on wake-up indicator
    DbncHi(BUTTON); // wait for stable button high
}

```

Complete program

The following listing shows how these fragments fit into the “wake-up on change demo” program:

```

/*****
*
* Description: Lesson 4, example 2
*
* Demonstrates wake-up on change
* plus differentiation from POR reset
*
* Turn on LED after each reset
* Turn on WAKE LED only if reset was due to wake on change
* then wait for button press, turn off LEDs, debounce, then sleep
*
*****
*
* Pin assignments:
* GP1 = on/off indicator LED
* GP2 = wake-on-change indicator LED
* GP3 = pushbutton switch (active low)
*
*****/

#include <12F509.h>

#define GP0 PIN_B0 // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5

#define GPWUF = 0x03.7 // GPWUF flag in STATUS register

#include "stdmacros-CCS.h" // DbnceHi() - debounce switch, wait for high
// Requires: TMR0 at 256us/tick

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC

// Pin assignments
#define LED GP1 // LED to turn on/off
#define WAKE GP2 // indicates wake on change condition
#define BUTTON GP3 // Pushbutton (active low)

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    output_b(0); // start with both LEDs off

    // configure wake-on-change and timer
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_256); // enable wake-up on change
                                                // configure Timer0:
                                                // timer mode, prescale = 256
                                                // -> increment every 256 us

```

```

//*** Main code

// turn on LED
output_high(LED);           // turn on LED

// test for wake-on-change reset
if (GPWUF)                 // if wake on change has occurred,
{
    output_high(WAKE);     // turn on wake-up indicator
    DbncHi(BUTTON);       // wait for button to stop bouncing
}

// wait for button press
while (input(BUTTON) == 1) // wait until button low
    ;

// go into standby (low power) mode
output_b(0);               // turn off both LEDs

DbncHi(BUTTON);           // wait for stable button release

sleep();                  // enter sleep mode
}

```

Watchdog Timer

As described in [baseline assembler lesson 7](#), the watchdog timer is free-running counter which, if enabled, operates independently of any program running on the PIC. It is typically used to avoid program crashes, where your application enters a state it will never return from, such as a loop waiting for a condition that will never occur. If the watchdog timer overflows, the PIC is reset, restarting your program – hopefully allowing it to recover and operate normally. To avoid this “WDT reset” from occurring, your program must periodically reset, or clear, the watchdog timer before it overflows. This watchdog time-out period on the baseline PICs is nominally 18 ms, but can be extended to a maximum of 2.3 seconds by assigning the prescaler to the watchdog timer (in which case the prescaler is no longer available for use with Timer0).

The watchdog timer can also be used to regularly wake the PIC from sleep mode, perhaps to sample and log an environmental input (say a temperature sensor), for low power operation.

The examples in this section illustrate these concepts.

Enabling the watchdog timer and detecting WDT resets

We saw in [baseline assembler lesson 7](#) that the watchdog timer is controlled by the WDTE bit in the processor configuration word: setting WDTE to ‘1’ enables the watchdog timer.

The assembler examples in that lesson included the following construct, to make it easy to select whether the watchdog timer is enabled or disabled when the code is built:

```

#define WATCHDOG           ; define to enable watchdog timer

IFDEF WATCHDOG
    __CONFIG                ; ext reset, no code protect, watchdog, int RC clock
    _MCLRE_ON & _CP_OFF & _WDT_ON & _IntRC_OSC
ELSE
    __CONFIG                ; ext reset, no code protect, no watchdog, int RC clock
    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC
ENDIF

```

To select the maximum watchdog time-out period of 2.3 seconds, the prescaler was assigned to the watchdog timer (by setting the PSA bit in OPTION), with a prescale ratio of 128:1 (18 ms × 128 = 2.3 s), by:

```
movlw    1<<PSA | b'111'      ; prescaler assigned to WDT (PSA = 1)
                                ; prescale = 128 (PS = 111)
option   ; -> WDT period = 2.3 s
```

To demonstrate the effect of the watchdog timer, an LED is turned on for 1 second, and then turned off, before the program enters an endless loop. Without the watchdog timer, the LED would remain off, until the power is cycled. But if the watchdog timer is enabled, a WDT reset will occur after 2.3 seconds, restarting the program, lighting the LED again. The LED will be seen to flash – on for 1 s, with a period of 2.3 s.

If you want your program to behave differently when restarted by a watchdog time-out, test the \overline{TO} flag in the STATUS register: it is cleared to '0' only when a WDT reset has occurred.

The example in [baseline assembler lesson 7](#) used this approach to turn on an “error” LED, to indicate if a restart was due to a WDT reset:

```
; test for WDT-timeout reset
btfss   STATUS,NOT_TO        ; if WDT timeout has occurred,
bsf     WDT                  ; turn on "error" LED

; flash LED
bsf     LED                  ; turn on "flash" LED
DelayMS 1000                 ; delay 1 sec
bcf     LED                  ; turn off "flash" LED

; wait forever
goto    $
```

XC8

Since the watchdog timer is controlled by a configuration bit, the only change we need to make to enable it is to use a different `__CONFIG()` statement, with the symbol 'WDT_ON' replacing 'WDT_OFF'.

A construct very similar to that in the assembler example can be used to select between processor configurations:

```
#define    WATCHDOG          // define to enable watchdog timer

#ifndef WATCHDOG
    // ext reset, no code protect, watchdog, int RC clock
    __CONFIG(MCLRE_ON & CP_OFF & WDT_ON & OSC_Intrc);
#else
    // ext reset, no code protect, no watchdog, int RC clock
    __CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & OSC_Intrc);
#endif
```

Assigning the prescaler to the watchdog timer and selecting a prescale ratio of 128:1 is done by:

```
OPTION = PSA | 0b111;        // prescaler assigned to WDT (PSA = 1)
                                // prescale = 128 (PS = 111)
                                // -> WDT period = 2.3 s
```

The symbol 'PSA' is defined in the header files provided with XC8.

To check for a WDT timeout reset, the $\overline{\text{TO}}$ flag can be tested directly, using:

```
if (!STATUSbits.nTO)           // if WDT timeout has occurred,
    WDT = 1;                   // turn on "error" LED
```

Note that the test condition is inverted, using ‘!’, since this flag is “active” when clear.

Complete program

Here is the complete program, showing how the above code fragments are used:

```

/*****
*   Description:    Lesson 4, example 3a
*
*   Demonstrates watchdog timer
*                   plus differentiation from POR reset
*
*   Turn on LED for 1 s, turn off, then enter endless loop
*   If enabled, WDT timer restarts after 2.3 s -> LED flashes
*   Turns on WDT LED to indicate WDT reset
*
*****/
*
*   Pin assignments:
*       GP1 = flashing LED
*       GP2 = WDT-reset indicator LED
*
*****/

#include <xc.h>

#define _XTAL_FREQ 4000000    // oscillator frequency for _delay()

/***** CONFIGURATION *****/
#define WATCHDOG              // define to enable watchdog timer

#ifdef WATCHDOG
    // ext reset, no code protect, watchdog, int RC clock
    __CONFIG(MCLRE_ON & CP_OFF & WDT_ON & OSC_IntrC);
#else
    // ext reset, no code protect, no watchdog, int RC clock
    __CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & OSC_IntrC);
#endif

// Pin assignments
#define LED      GPIObits.GP1    // LED to flash
#define nLED    1                // (port bit 1)
#define WDT     GPIObits.GP2    // watchdog timer reset indicator
#define nWDT    2                // (port bit 2)

/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    GPIO = 0;                // start with all LEDs off
    TRIS = ~(1<<nLED|1<<nWDT); // configure LED pins as outputs
}

```

```

// configure watchdog timer
OPTION = PSA | 0b111;          // prescaler assigned to WDT (PSA = 1)
                                // prescale = 128 (PS = 111)
                                // -> WDT period = 2.3 s

/** Main code
// test for WDT-timeout reset
if (!STATUSbits.nTO)         // if WDT timeout has occurred,
    WDT = 1;                 // turn on "error" LED

// flash LED
LED = 1;                     // turn on "flash" LED
__delay_ms(1000);           // delay 1 sec
LED = 0;                     // turn off "flash" LED

// wait forever
for (;;)
    ;
}

```

CCS PCB

To enable the watchdog timer, simply replace the symbol 'NOWDT' with 'WDT' in the #fuses statement.

Once again, we can use a conditional compilation construct to allow the watchdog to be enabled or disabled when building the code:

```

#define WATCHDOG              // define to enable watchdog timer

#ifndef WATCHDOG
    // ext reset, no code protect, watchdog, int RC clock
    #fuses MCLR,NOPROTECT,WDT,INTRC
#else
    // ext reset, no code protect, no watchdog, int RC clock
    #fuses MCLR,NOPROTECT,NOWDT,INTRC
#endif

```

Unlike the situation for enabling and detecting wake-up on change, the CCS PCB compiler provides built-in functions for setting up the watchdog timer and detecting that a WDT reset has occurred.

Although it is possible to use the 'setup_counters()' function to setup the watchdog timer, CCS has de-emphasised its use, in favour of the more specific 'setup_wdt()'.

The setup_wdt() function takes a single parameter, which on the baseline PICs specifies the watchdog timeout period, from 'WDT_18MS' (18 ms), 'WDT_36MS' (36 ms), 'WDT_72MS' (72 ms), etc., through to 'WDT_2304MS' (2.3 s).

So in this example we have:

```

setup_wdt(WDT_2304MS);          // WDT period = 2.3 s)

```

As mentioned above, one of the available built-in functions is 'restart_cause()', which returns a value indicating why the PIC was (re)started. Although it doesn't accommodate wake-up on change resets, it does correctly detect WDT resets, in which case it returns the value corresponding to 'WDT_TIMEOUT' (a symbol defined in the "12F509.h" header file). For example:

```

if (restart_cause() == WDT_TIMEOUT) // if WDT timeout has occurred,
    output_high(WDT);              // turn on "error" LED

```

There is, however, one complicating factor: `setup_wdt()` has the side effect of resetting the \overline{TO} flag, which the `restart_cause()` function relies on to determine whether a WDT timeout had occurred.

That is, if `setup_wdt()` is called before `restart_cause()`, the information about why the restart had happened is lost. Therefore, it is important to call `restart_cause()` before `setup_wdt()`, as in the following program.

Complete program

Here is how the code fits together, when using CCS PCB:

```

/*****
 *
 * Description: Lesson 4, example 3a
 *
 * Demonstrates watchdog timer
 * plus differentiation from POR reset
 *
 * Turn on LED for 1 s, turn off, then enter endless loop
 * If enabled, WDT timer restarts after 2.3 s -> LED flashes
 * Turns on WDT LED to indicate WDT reset
 *
 *****/
 *
 * Pin assignments:
 * GP1 = flashing LED
 * GP2 = WDT-reset indicator LED
 *
 *****/
#include <12F509.h>

#define GP0 PIN_B0 // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5

#define delay (clock=4000000) // oscillator frequency for delay_ms()

/***** CONFIGURATION *****/
#define WATCHDOG // define to enable watchdog timer

#ifdef WATCHDOG
// ext reset, no code protect, watchdog, int RC clock
#fuses MCLR,NOPROTECT,WDT,INTRC
#else
// ext reset, no code protect, no watchdog, int RC clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC
#endif

// Pin assignments
#define LED GP1 // LED to flash
#define WDT GP2 // watchdog timer reset indicator

/***** MAIN PROGRAM *****/
void main()
{
//***** Initialisation

```

```

// configure port
output_b(0); // start with both LEDs off

// test for WDT-timeout reset
// (note: must be done before initialising watchdog timer)
if (restart_cause() == WDT_TIMEOUT) // if WDT timeout has occurred,
    output_high(WDT); // turn on "error" LED

// configure watchdog timer
setup_wdt(WDT_2304MS); // WDT period = 2.3 s

//***** Main code
// flash LED
output_high(LED); // turn on "flash" LED
delay_ms(1000); // delay 1 sec
output_low(LED); // turn off "flash" LED

// wait forever
while (TRUE)
    ;
}

```

Clearing the watchdog timer

The previous example shows what happens when the watchdog timer overflows, but of course most of the time, during “normal” program operation, we want to prevent that from happening; a WDT reset should only occur when something has gone wrong.

As mentioned above, to avoid overflows, the watchdog timer has to be regularly cleared. This is typically done by inserting a ‘clrwdt’ instruction within the program’s “main loop”, and within any subroutine which may, in normal operation, not complete within the watchdog timer period.

To demonstrate the effect of clearing the watchdog timer, a ‘clrwdt’ instruction was added into the endless loop in the example in [baseline assembler lesson 7](#):

```

;***** Main code
    bsf      LED           ; turn on LED

    DelayMS 1000         ; delay 1 sec

    bcf      LED           ; turn off LED

loop   clrwdt            ; clear watchdog timer
      goto   loop        ; repeat forever

```

With the ‘clrwdt’ instruction in place, the watchdog timer never overflows, so the PIC is never restarted by a WDT reset, and the LED remains turned off until the power is cycled, whether the watchdog timer is enabled or not.

XC8

Similar to the ‘SLEEP()’ macro we saw earlier, XC8 provides a ‘CLRWDT()’ macro, defined in the “pic.h” header file as:

```
#define CLRWDT() asm("clrwdt")
```

That is, the ‘CLRWDT()’ macro simply inserts a ‘clrwdt’ instruction into the code.

Using this macro, the assembler example above can be implemented with XC8 as follows:

```

/***/ Main code
LED = 1;                // turn on LED

__delay_ms(1000);      // delay 1 sec

LED = 0;                // turn off LED

for (;;)                // repeatedly clear watchdog timer forever
    CLRWDT();

```

CCS PCB

Instead of a macro, the CCS PCB compiler provides a built-in function for clearing the watchdog timer:

```
restart_wdt();
```

Here is the CCS PCB code, equivalent to the example above, using the `restart_wdt()` function:

```

/***/ Main code
output_high(LED);      // turn on LED

delay_ms(1000);        // delay 1 sec

output_low(LED);       // turn off LED

while (TRUE)           // repeatedly clear watchdog timer forever
    restart_wdt();

```

Periodic wake from sleep

As explained in [baseline assembler lesson 7](#), the watchdog timer is also often used to periodically wake the PIC from sleep mode, typically to check or log some inputs, take some action and then return to sleep mode, saving power. This can be combined with wake-up on pin change, allowing immediate response to some inputs, such as a button press, while periodically checking others.

To illustrate this, the example in that lesson replaced the endless loop with a ‘sleep’ instruction:

```

;***** Main code
    bsf      LED                ; turn on LED

    DelayMS 1000                ; delay 1 sec

    bcf      LED                ; turn off LED

    sleep                       ; enter sleep mode

```

With the watchdog timer enabled, with a period of 2.3 s, the LED is on for 1 s, and then off for 1.3 s, as in the earlier example. But this time the PIC is in sleep mode while the LED is off, conserving power.

XC8

There are no new instructions or concepts needed for this example; the main code is simply:

```

//*** Main code
LED = 1;                // turn on LED

__delay_ms(1000);      // delay 1 sec

LED = 0;                // turn off LED

SLEEP();               // enter sleep mode

```

CCS PCB

Again, there are no new statements needed; the main code is much the same as we have seen before:

```

//*** Main code
output_high(LED);      // turn on LED

delay_ms(1000);        // delay 1 sec

output_low(LED);       // turn off LED

sleep();               // enter sleep mode

```

Clock (Oscillator) Options

[Baseline lesson 7](#) also discussed the various clock, or oscillator, configurations available on the PIC12F509.

A number of examples were used to demonstrate the various options. Since the only new features in these examples were the configuration word settings, and no other new concepts were introduced, there would be little point in reproducing C versions of those examples here.

However, for reference, here is a summary of the oscillator configuration options for the XC8 and CCS compilers, with the corresponding MPASM symbols:

FOSC<1:0>	Oscillator configuration	MPASM	XC8	CCS PCB
00	LP oscillator	<code>_LP_OSC</code>	<code>OSC_LP</code>	LP
01	XT oscillator	<code>_XT_OSC</code>	<code>OSC_XT</code>	XT
10	Internal RC oscillator	<code>_IntRC_OSC</code>	<code>OSC_IntRC</code>	INTRC
11	External RC oscillator	<code>_ExtRC_OSC</code>	<code>OSC_ExtRC</code>	RC

For example, to configure the processor for use with a LP crystal using XC8, you could use:

```

// ext reset, no code protect, watchdog, LP crystal
__CONFIG(MCLRE_ON & CP_OFF & WDT_ON & OSC_LP);

```

Or to set the processor configuration for an external RC oscillator using CCS PCB, you could use:

```

// ext reset, no code protect, watchdog, ext RC oscillator
#fuses MCLR,NOPROTECT,WDT,RC

```

Summary

Overall, we have seen that the sleep mode, wake-up on change, and watchdog timer features of the baseline PIC architecture can be accessed effectively in C programs, using both the XC8 and CCS compilers.

However, CCS PCB lacks support for detecting wake-on-change resets, and its watchdog timer setup function has a side effect which meant that we had to rearrange one example. This (non-obvious side effects) is certainly a potential issue when using compilers such as CCS', which hide the details of the PIC architecture behind built-in functions. On the other hand, the CCS source code in all of the examples is concise and clear. You just need to be aware of the potential for side effects from those functions.

The [next lesson](#) will focus on driving 7-segment displays (revisiting the material from [baseline assembler lesson 8](#)), showing how lookup tables and multiplexing can be implemented using C.

And to do that, we'll introduce the 14-pin PIC16F506.

Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 5: Driving 7-Segment Displays

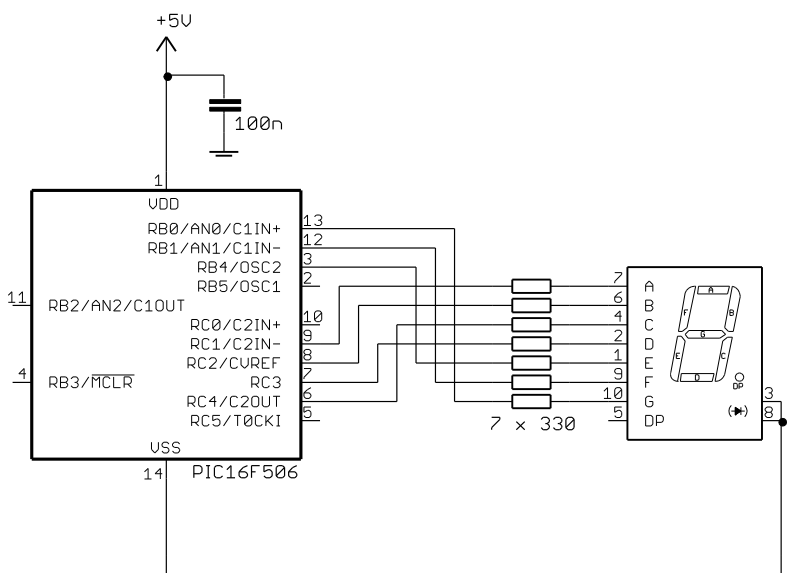
We saw in [baseline assembler lesson 8](#) how to drive 7-segment LED displays, using lookup-tables and multiplexing techniques implemented in assembly language. This lesson shows how C can be used to apply those techniques to drive multiple 7-segment displays, using the Microchip's XC8 (running in "Free mode") and CCS' PCB¹ compilers to re-implement the examples.

In summary, this lesson covers:

- Using lookup tables to drive a single 7-segment display
- Using multiplexing to drive multiple displays

Lookup Tables and 7-Segment Displays

To demonstrate how to drive a single 7-segment display, we will use the circuit from [baseline assembler lesson 8](#), as shown below.



It uses a 16F506 which, as was explained in that lesson, is a 14-pin baseline PIC, with analog inputs (comparators and ADC), more oscillator modes and more data memory, but is otherwise similar to the 12F509 used in the earlier lessons. It provides two 6-pin ports: PORTB and PORTC.

A common-cathode 7-segment LED module is used here. The common-cathode connection is grounded. Each segment is driven, via a 330 Ω resistor, directly from one of the output pins. To light a given segment, the corresponding output is set high.

If a common-anode module is used instead, the anode connection is connected to VDD and the pins become active-low (cleared to zero to make the connected segment light) – you would need to make appropriate changes to the examples below.

¹ XC8 is available as a free download from www.microchip.com, and CCS PCB is bundled for free with MPLAB 8

If you are using the [Gooligum baseline training board](#), you can implement this circuit by:

- placing shunts (six of them) across every position in jumper block JP4, connecting segments A-D, F and G to pins RB0-1 and RC1-4
- placing a single shunt in position 1 (“RA/RB4”) of JP5, connecting segment E to pin RB4
- placing a shunt across pins 1 and 2 (“GND”) of JP6, connecting digit 1 to ground.

All other shunts should be removed.

As we saw in [baseline assembler lesson 8](#), lookup tables on baseline PICs are normally implemented as a computed jump into a sequence of ‘retlw’ instructions, each returning a value corresponding to its position in the table. Care has to be taken to ensure that the table is wholly contained within the first 256 words of a program memory page, and that the page selection bits are set correctly before accessing (calling) the table.

The example program in that lesson implemented a simple seconds counter, displaying each digit from 0 to 9, then repeating, with a 1 s delay between each count.

XC8

In C, a lookup table would usually be implemented as an initialised array. For example:

```
uint8_t days[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

The problem with such a declaration for XC8 is that the compiler has no way to know whether the array contents will change, so it is forced to place such an array in data memory (which even in larger 8-bit PICs is a very limited resource) and add code to initialise the array on program start-up – wasteful of both data and program space.

If, instead, the array is declared as ‘const’, the compiler knows that the contents of the array will never change, and so can be placed in ROM (program memory), as a lookup table of retlw instructions.

So to create lookup tables equivalent to those in the assembler example in [baseline lesson 8](#), we can write:

```
// pattern table for 7 segment display on port B
const uint8_t pat7segB[10] = {
    // RB4 = E, RB1:0 = FG
    0b010010, // 0
    0b000000, // 1
    0b010001, // 2
    0b000001, // 3
    0b000011, // 4
    0b000011, // 5
    0b010011, // 6
    0b000000, // 7
    0b010011, // 8
    0b000011 // 9
};

// pattern table for 7 segment display on port C
const uint8_t pat7segC[10] = {
    // RC4:1 = CDBA
    0b011110, // 0
    0b010100, // 1
    0b001110, // 2
    0b011110, // 3
    0b010100, // 4
    0b011010, // 5
    0b011010, // 6
    0b010110, // 7
};
```

```

    0b0111110,    // 8
    0b0111110    // 9
};

```

Looking up the display patterns is easy; the digit to be displayed is used as the array index.

To set the port pins for a given digit, we then have:

```

    PORTB = pat7segB[digit];    // lookup port B and C patterns
    PORTC = pat7segC[digit];

```

This is quite straightforward, and certainly simpler than the assembler version.

However, the assembler example used two tables, one for **PORTB**, the other for **PORTC**, to simplify the code for writing the appropriate pattern to each port. In C, it is easier to write more complex expressions, without having to be as concerned by (or even aware of) such implementation details.

In this case, if you were writing the C program for this example from scratch, instead of converting an existing assembler program, it may seem more natural to use a single lookup table with patterns specifying all seven segments of the display, and to then extract the parts of each pattern corresponding to various pins.

For example:

```

// pattern table for 7 segment display on ports B and C
const uint8_t pat7seg[10] = {
    // RC4:1, RB4, RB1:0 = CDBAEFG
    0b1111110,    // 0
    0b1010000,    // 1
    0b0111101,    // 2
    0b1111001,    // 3
    0b1010011,    // 4
    0b1101011,    // 5
    0b1101111,    // 6
    0b1011000,    // 7
    0b1111111,    // 8
    0b1111011    // 9
};

```

Bits 6:3 of each pattern provide the **PORTC** bits 4:1, so to get the value for **PORTC**, shift the pattern two bits to the right, and mask off bit 0:

```

    PORTC = (pat7seg[digit] >> 2) & 0b0111110;

```

Extracting the bits for **PORTB** is a little more difficult.

Pattern bit 2 gives the value for **RB4**. To extract that bit (by ANDing with a single-bit mask) and shift it to position 4 (corresponding to **RB4**), we can use the expression:

```

    (pat7seg[digit] & 1<<2) << 2

```

Pattern bits 1:0 give the values of **PORTB** bits 1:0 (**RB1** and **RB0**). We don't need to do any shifting; the bit positions already align, so to extract these bits, we can simply AND them with a mask:

```

    (pat7seg[digit] & 0b00000011)

```

Finally, we need to **OR** these two expressions together, to build the value to load into **PORTB**:

```

    PORTB = (pat7seg[digit] & 1<<2) << 2 |
            (pat7seg[digit] & 0b00000011);

```

Whether you would choose to do this in practice (it seems a bit clumsy here) is partly a matter of personal style, and also a question of whether the space savings, from using only one pattern array, are worth it.

Because we are now using a PIC16F506, instead of the simpler 12F509, there are a couple of differences from our earlier XC8 programs, in configuration and port initialisation, to be aware of.

The 16F506 includes an analog-to-digital converter and two analog comparators. As explained in [baseline assembler lesson 8](#), the analog inputs associated with these peripherals must be disabled before those pins can be used for digital I/O, and this can be done by clearing the ADCON0 register (to deselect all of the ADC inputs) and the C1ON and C2ON bits (to disable the two comparators).

This can be done in XC8 by:

```
ADCON0 = 0;           // disable AN0, AN1, AN2 inputs
CM1CON0bits.C1ON = 0; // and comparator 1 -> RB0,RB1 digital
CM2CON0bits.C2ON = 0; // disable comparator 2 -> RC1 digital
```

We also saw that the 16F506 supports a wider range of clock options than the 12F509. Since we want to use the internal RC oscillator, with RB4 available for I/O, we need to use the 'OSC_IntRC_RB4EN' symbol, instead of 'OSC_IntRC', and include 'IOSCFS_OFF' (to configure the internal oscillator for 4 MHz operation) in the __CONFIG() macro, as follows:

```
// ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & IOSCFS_OFF & OSC_IntRC_RB4EN);
```

For the full list of configuration symbols for the 16F506, see the "pic16f506.h" file in the XC8 include directory.

Complete program

Here is the complete single-lookup-table version of this example, for XC8:

```
/******
 * Description: Lesson 5, example 1b
 *
 * Demonstrates use of lookup tables to drive a 7-segment display
 *
 * Single digit 7-segment display counts repeating 0 -> 9
 * 1 second per count, with timing derived from int 4 MHz oscillator
 * (single pattern lookup array)
 *
 *
 * Pin assignments:
 * RB0-1,RB4, RC1-4 = 7-segment display bus (common cathode)
 *
 *****/

#include <xc.h>
#include <stdint.h>

#define _XTAL_FREQ 4000000 // oscillator frequency for delay functions

/****** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & IOSCFS_OFF & OSC_IntRC_RB4EN);

/****** LOOKUP TABLES *****/

// pattern table for 7 segment display on ports B and C
const uint8_t pat7seg[10] = {
    // RC4:1,RB4,RB1:0 = CDBAEFG
    0b1111110, // 0
    0b1010000, // 1
```

```

    0b0111101, // 2
    0b1111001, // 3
    0b1010011, // 4
    0b1101011, // 5
    0b1101111, // 6
    0b1011000, // 7
    0b1111111, // 8
    0b1111011 // 9
};

/***** MAIN PROGRAM *****/
void main()
{
    uint8_t    digit;           // digit to be displayed

    /*** Initialisation

    // configure ports
    TRISB = 0;                  // configure PORTB and PORTC as all outputs
    TRISC = 0;
    ADCON0 = 0;                 // disable AN0, AN1, AN2 inputs
    CM1CON0bits.C1ON = 0;       // and comparator 1 -> RB0,RB1 digital
    CM2CON0bits.C2ON = 0;       // disable comparator 2 -> RC1 digital

    /*** Main loop
    for (;;)
    {
        // display each digit from 0 to 9 for 1 sec
        for (digit = 0; digit < 10; digit++)
        {
            // display digit by extracting pattern bits for all pins
            PORTB = (pat7seg[digit] & 1<<2) << 2 |           // RB4
                    (pat7seg[digit] & 0b00000011);           // RB0-1
            PORTC = (pat7seg[digit] >> 2) & 0b0111110;         // RC1-4

            // delay 1 sec
            __delay_ms(1000);
        }
    }
}

```

CCS PCB

Like XC8, the CCS PCB compiler also places initialised arrays in program memory, as a table of `retlw` instructions, if the array is declared with the `'const'` qualifier.

Hence, the pattern lookup array is defined in the same way as for XC8.

The expressions for extracting the pattern bits are also the same, since they are standard ANSI syntax. But of course, the statements for assigning those patterns to the port pins are different, because CCS PCB uses built-in functions:

```

    output_b((pat7seg[digit] & 1<<2) << 2 |           // RB4
             (pat7seg[digit] & 0b00000011));         // RB0-1
    output_c((pat7seg[digit] >> 2) & 0b0111110);     // RC1-4

```

As we did in the XC8 version, we need to make some changes to the configuration and port initialisation code, to reflect the fact that we're using a 16F506 instead of a 12F509.

To disable the analog inputs, making all pins available for digital I/O, we can use the built-in functions `setup_comparator()` and `setup_adc_ports()`. We'll see how to use them in lessons [6](#) and [7](#), but for now we can simply use:

```
setup_adc_ports(NO_ANALOGS);    // disable all analog and comparator inputs
setup_comparator(NC_NC_NC_NC); // -> RB0, RB1, RC0, RC1 digital
```

We also need to update the `#fuses` statement to use the internal RC oscillator, with RB4 available for I/O, by using the 'INTRC_IO' symbol instead of 'INTRC', and to run at 4 MHz, by including the symbol 'IOSC4', as follows:

```
// ext reset, no code protect, no watchdog, 4 MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,IOSC4,INTRC_IO
```

Note also that to define these symbols, you must include the correct header file for the target PIC – in this case it is "16F506.h" (located in the CCS PCB "Devices" directory), where you will find the full list of configuration symbols for the 16F506.

Finally, now that we're using a device with a port B, there is no need for the `#define` statements we used in the 12F509 examples to define GP pin labels.

Complete program

Here is the complete single-table-lookup version of the program, for CCS PCB:

```

/*****
*
* Description: Lesson 5, example 1b
*
* Demonstrates use of lookup tables to drive a 7-segment display
*
* Single digit 7-segment display counts repeating 0 -> 9
* 1 second per count, with timing derived from int 4 MHz oscillator
* (single pattern lookup array)
*
*****/
*
* Pin assignments:
* RB0-1, RB4, RC1-4 = 7-segment display bus (common cathode)
*
*****/

#include <16F506.h>

#use delay (clock=4000000) // oscillator frequency for delay_ms()

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,IOSC4,INTRC_IO

/***** LOOKUP TABLES *****/

// pattern table for 7 segment display on ports B and C
const int8 pat7seg[10] = {
    // RC4:1, RB4, RB1:0 = CDBAEFG
    0b1111110, // 0
    0b1010000, // 1
    0b0111101, // 2
    0b1111001, // 3
    0b1010011, // 4

```

```

    0b1101011, // 5
    0b1101111, // 6
    0b1011000, // 7
    0b1111111, // 8
    0b1111011 // 9
};

/***** MAIN PROGRAM *****/
void main()
{
    unsigned int8    digit;           // digit to be displayed

    /*** Initialisation
    // configure ports
    setup_adc_ports(NO_ANALOGS);     // disable all analog and comparator inputs
    setup_comparator(NC_NC_NC_NC);   // -> RB0, RB1, RC0, RC1 digital

    /*** Main loop
    while (TRUE)
    {
        // display each digit from 0 to 9 for 1 sec
        for (digit = 0; digit < 10; digit++)
        {
            // display digit by extracting pattern bits for all pins
            output_b((pat7seg[digit] & 1<<2) << 2 |           // RB4
                    (pat7seg[digit] & 0b00000011));           // RB0-1
            output_c((pat7seg[digit] >> 2) & 0b011110);       // RC1-4

            // delay 1 sec
            delay_ms(1000);
        }
    }
}

```

Comparisons

The following table summarises the source code length and resource usage for the “single-digit seconds counter” assembly and C example programs, for the versions (assembly and C) with two lookup tables with direct port updates, and the C versions that use a single combined lookup array with more complex pattern extraction for each port.

Count_7seg_x1

Assembler / Compiler	Lookup tables	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	per-port	67	73	4
XC8 (Free mode)	per-port	38	104	4
CCS PCB	per-port	34	92	7
XC8 (Free mode)	combined	27	125	4
CCS PCB	combined	23	98	10

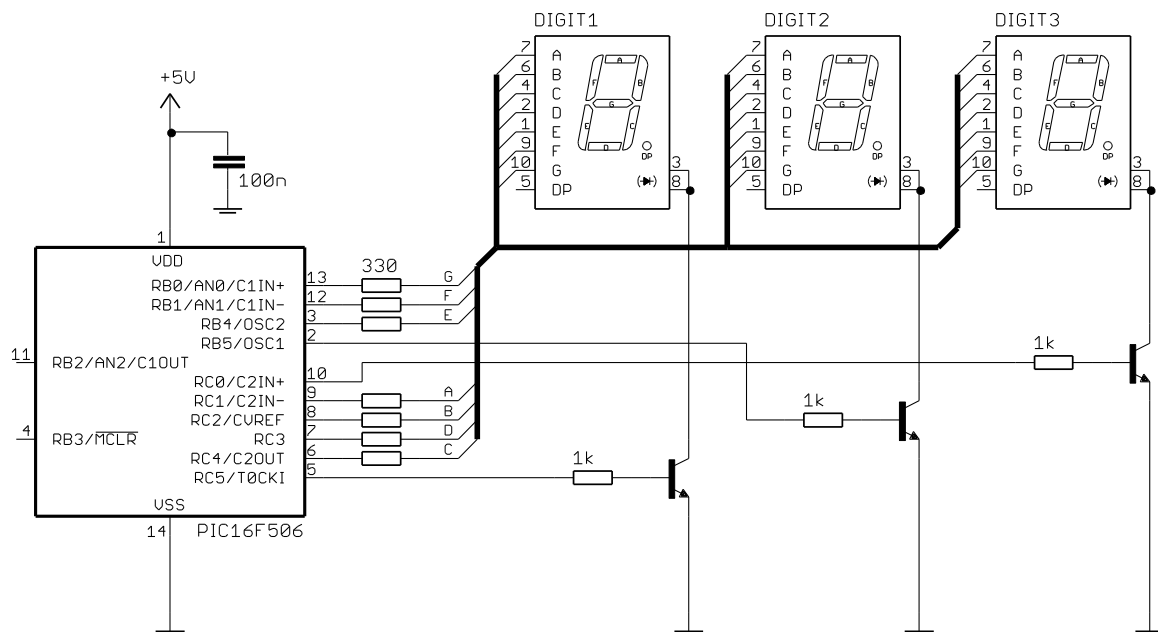
As you can see, the table-per-port C versions are much shorter than the assembler equivalent – and the versions using a single combined lookup table are even shorter. But even with only one table in memory, the C compilers still generate larger code than the two-table version – due to the instructions needed to extract the patterns from each array entry.

In this case, the added complexity of the code needed to extract bit patterns from an combined lookup array isn't worth it – the generated code becomes bigger overall, and the CCS compiler used a lot of extra data memory (presumably to store intermediate results during the extraction process). And the combined-table version is arguably harder to understand. Nevertheless, if the lookup tables were much longer (say 50 entries instead of 10), it would be a different story – the space saved by storing only a single table in memory would more than make up for the extra instructions needed to decode it. Sometimes you simply need to try both ways, to see what's best.

Multiplexing

As explained in more detail in [baseline assembler lesson 8](#), *multiplexing* can be used to drive multiple displays, using a minimal number of output pins. Each display is lit in turn, one at a time, so rapidly that it appears to the human eye that each display is lit continuously.

We'll use the example circuit from that lesson, shown below, to demonstrate how to implement this technique, using C.



To implement this circuit using the [Gooligum baseline training board](#):

- keep the six shunts in every position of jumper block JP4, connecting segments A-D, F and G to pins RB0-1 and RC1-4
- keep the shunt in position 1 (“RA/RB4”) of JP5, connecting segment E to pin RB4
- move the shunt in JP6 to across pins 2 and 3 (“RC5”), connecting digit 1 to the transistor controlled by RC5
- place shunts in jumpers JP8, JP9 and JP10, connecting pins RC5, RB5 and RC0 to their respective transistors

All other shunts should be removed.

Each 7-segment display is enabled (able to be lit when its segment inputs are set high) when the NPN transistor connected to its cathode pins is turned on (by pulling the base high), providing a path to ground.

To multiplex the display, each transistor is turned on (by setting high the pin connected to its base) in turn, while outputting the pattern corresponding to that digit on the segment pins, which are wired as a bus.

To ensure that the displays are lit evenly, a timer should be used to ensure that each display is enabled for the same period of time. In the assembler example, this was done as follows:

```

        ; display minutes for 2.048 ms
w60_hi  btfss   TMR0,2           ; wait for TMR<2> to go high
        goto   w60_hi
        movf   mins,w           ; output minutes digit
        pagesel set7seg
        call   set7seg
        pagesel $
w60_lo  bsf    MINUTES         ; enable minutes display
        btfsc  TMR0,2           ; wait for TMR<2> to go low
        goto   w60_lo

```

Timer0 is used to time the display sequencing; it is configured such that bit 2 cycles every 2.048 ms, providing a regular *tick* to base the multiplex timing on.

Since each display is enabled for 2.048 ms, and there are three displays, the output is refreshed every 6.144 ms, or about 162 times per second – fast enough to appear continuous.

The assembler example implemented a minutes and seconds counter, so the output refresh process was repeated for 1 second (i.e. 162 times), before incrementing the count.

This approach is not 100% accurate (the prototype had a measured accuracy of 0.3% over ten minutes), but given that the timing is based on the internal RC oscillator, which is only accurate to within 1% or so, that's not really a problem.

XC8

In the assembler version of this example ([baseline lesson 8](#), example 2), the time count digits were stored as a separate variables:

```

        UDATA
mins    res 1           ; current count: minutes
tens    res 1           ; tens
ones    res 1           ; ones

```

This was done to simplify the assembler code, which, at the end of the main loop, incremented the “ones” variable, and if it overflowed from 9 to 0, incremented “tens” (and on a “tens” overflow from 5 to 0, incremented “minutes”).

The next example ([baseline lesson 8](#), example 3) then showed how the seconds value could be stored in a single value, using BCD format to simplify the process of extracting each digit for display:

```

        UDATA
mins    res 1           ; time count: minutes
secs    res 1           ; seconds (BCD)

```

For example, to extract and display the tens digit, we had:

```

        swapf   secs,w           ; get tens digit
        andlw   0x0F           ; from high nybble of seconds
        pagesel set7seg
        call    set7seg         ; then output it

```


However, in C it is far more natural to simply store minutes and seconds as ordinary integer variables:

```
uint8_t    mins, secs;           // time counters
```

And then the tens digit would be extracted by dividing seconds by ten, and displayed, as follows:

```
PORTB = pat7segB[secs/10];      // output tens digit
PORTC = pat7segC[secs/10];      //   on display bus
```

Similarly, the ones digit is returned by the simple expression 'secs%10', which gives the remainder after dividing seconds by ten.

Of course we need some code round that, to wait for TMR0<2> to go high and then low, and to enable the appropriate display module:

```
// display tens for 2.048 ms
while (!(TMR0 & 1<<2))          // wait for TMR0<2> to go high
    ;
PORTB = 0;                       // disable displays
PORTC = 0;
PORTB = pat7segB[secs/10];       // output tens digit
PORTC = pat7segC[secs/10];       //   on display bus
TENS = 1;                         // enable tens display only
while (TMR0 & 1<<2)              // wait for TMR0<2> to go low
    ;
```

This code assumes that the symbol 'TENS' has been defined:

```
// Pin assignments
#define MINUTES PORTCbits.RC5     // minutes enable
#define TENS     PORTBbits.RB5     // tens enable
#define ONES     PORTCbits.RC0     // ones enable
```

The block of code to display the tens digit has to be repeated with only minor variations for the minutes and ones digits.

This repetition can be reduced in a couple of ways.

The expression 'TMR0 & 1<<2', used to access TMR0<2>, is a little unwieldy. Since it is used six times in the program (twice for each digit), it makes sense to define it as a macro:

```
#define TMR0_2 (TMR0 & 1<<2)     // access to TMR0<2>
```

The loop which waits for TMR0<2> to go high can then be written more simply as:

```
while (!TMR0_2)                  // wait for TMR0<2> to go high
    ;
```

and to wait for it to go low:

```
while (TMR0_2)                   // wait for TMR0<2> to go low
    ;
```

More significantly, the code which outputs the digit patterns can be implemented as a function:

```
void set7seg(uint8_t digit)
{
    // pattern table for 7 segment display on port B
    const uint8_t pat7segB[10] = {
        // RB4 = E, RB1:0 = FG
        0b010010, // 0
        0b000000, // 1
        0b010001, // 2
        0b000001, // 3
        0b000011, // 4
        0b000011, // 5
        0b010011, // 6
        0b000000, // 7
        0b010011, // 8
        0b000011 // 9
    };

    // pattern table for 7 segment display on port C
    const uint8_t pat7segC[10] = {
        // RC4:1 = CDBA
        0b011110, // 0
        0b010100, // 1
        0b001110, // 2
        0b011110, // 3
        0b010100, // 4
        0b011010, // 5
        0b011010, // 6
        0b010110, // 7
        0b011110, // 8
        0b011110 // 9
    };

    // Disable displays
    PORTB = 0; // clear all digit enable lines on PORTB
    PORTC = 0; // and PORTC

    // Output digit pattern
    PORTB = pat7segB[digit]; // lookup and output port B and C patterns
    PORTC = pat7segC[digit];
}

```

It makes sense to include the pattern table definition within the function, so that the function is self-contained – only the function needs to “know” about the pattern table; it is never accessed directly from other parts of the program. This is very similar to what was done in the assembler examples.

It also makes sense to include the code to disable the displays, prior to outputting a new pattern on the segment bus, within this function, since otherwise it would have to be repeated for each digit.

Displaying the tens digit then becomes:

```

// display tens for 2.048 ms
while (!TMR0_2) // wait for TMR0<2> to go high
    ;
set7seg(secs/10); // output tens digit
TENS = 1; // enable tens display
while (TMR0_2) // wait for TMR0<2> to go low
    ;

```

This is much more concise than before.

To display all three digits of the current count for 1 second, we then have:

```
// for each time count, multiplex display for 1 second
// (display each of 3 digits for 2.048 ms each,
// so repeat 1000000/2048/3 times to make 1 second)
for (mpx_cnt = 0; mpx_cnt < 1000000/2048/3; mpx_cnt++)
{
    // display minutes for 2.048 ms
    while (!TMR0_2)          // wait for TMR0<2> to go high
        ;
    set7seg(mins);          // output minutes digit
    MINUTES = 1;           // enable minutes display
    while (TMR0_2)          // wait for TMR0<2> to go low
        ;

    // display tens for 2.048 ms
    while (!TMR0_2)          // wait for TMR0<2> to go high
        ;
    set7seg(secs/10);       // output tens digit
    TENS = 1;               // enable tens display
    while (TMR0_2)          // wait for TMR0<2> to go low
        ;

    // display ones for 2.048 ms
    while (!TMR0_2)          // wait for TMR0<2> to go high
        ;
    set7seg(secs%10);       // output ones digit
    ONES = 1;               // enable ones display
    while (TMR0_2)          // wait for TMR0<2> to go low
        ;
}
}
```

Finally, instead of taking the assembler approach of incrementing all the counters (checking for and reacting to overflows) at the end of an endless loop, it seems much more natural in C to use nested `for` loops:

```
/***/ Main loop
for (;;)
{
    // count in seconds from 0:00 to 9:59
    for (mins = 0; mins < 10; mins++)
    {
        for (secs = 0; secs < 60; secs++)
        {
            // for each time count, multiplex display for 1 second

            // display multiplexing loop goes here
        }
    }
}
}
```

Complete program

Fitting all this together, including function prototypes, we have:

```
/*****
*
* Description:      Lesson 5, example 2
*
* Demonstrates use of multiplexing to drive multiple 7-seg displays
*
***/
```

```

*   3-digit 7-segment LED display: 1 digit minutes, 2 digit seconds   *
*   counts in seconds 0:00 to 9:59 then repeats,                     *
*   with timing derived from int 4 MHz oscillator                     *
*                                                                 *
*****
*
*   Pin assignments:
*       RB0-1, RB4, RC1-4 = 7-segment display bus (common cathode)
*       RC5                = minutes enable (active high)
*       RB5                = tens enable
*       RC0                = ones enable
*
*****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & IOSCF5_OFF & OSC_IntRC_RB4EN);

// Pin assignments
#define MINUTES PORTCbits.RC5    // minutes enable
#define TENS     PORTBbits.RB5   // tens enable
#define ONES     PORTCbits.RC0   // ones enable

/***** PROTOTYPES *****/
void set7seg(uint8_t digit);    // display digit on 7-segment display

/***** MACROS *****/
#define TMR0_2 (TMR0 & 1<<2)  // access to TMR0<2>

/***** MAIN PROGRAM *****/
void main()
{
    uint8_t    mpx_cnt;          // multiplex counter
    uint8_t    mins, secs;      // time counters

    /*** Initialisation

    // configure ports
    TRISB = 0;                  // configure PORTB and PORTC as all outputs
    TRISC = 0;
    ADCON0 = 0;                 // disable AN0, AN1, AN2 inputs
    CM1CON0bits.C1ON = 0;       // and comparator 1 -> RB0, RB1 digital
    CM2CON0bits.C2ON = 0;       // disable comparator 2 -> RC1 digital

    // configure timer
    OPTION = 0b11010111;        // configure Timer0:
        //--0-----           timer mode (T0CS = 0) -> RC5 usable
        //----0---             prescaler assigned to Timer0 (PSA = 0)
        //-----111           prescale = 256 (PS = 111)
        //                    -> increment every 256 us
        //                    (TMR0<2> cycles every 2.048 ms)

```

```

//*** Main loop
for (;;)
{
    // count in seconds from 0:00 to 9:59
    for (mins = 0; mins < 10; mins++)
    {
        for (secs = 0; secs < 60; secs++)
        {
            // for each time count, multiplex display for 1 second
            // (display each of 3 digits for 2.048 ms each,
            // so repeat 1000000/2048/3 times to make 1 second)
            for (mpx_cnt = 0; mpx_cnt < 1000000/2048/3; mpx_cnt++)
            {
                // display minutes for 2.048 ms
                while (!TMR0_2)          // wait for TMR0<2> to go high
                ;
                set7seg(mins);           // output minutes digit
                MINUTES = 1;             // enable minutes display
                while (TMR0_2)          // wait for TMR0<2> to go low
                ;

                // display tens for 2.048 ms
                while (!TMR0_2)          // wait for TMR0<2> to go high
                ;
                set7seg(secs/10);        // output tens digit
                TENS = 1;                // enable tens display
                while (TMR0_2)          // wait for TMR0<2> to go low
                ;

                // display ones for 2.048 ms
                while (!TMR0_2)          // wait for TMR0<2> to go high
                ;
                set7seg(secs%10);        // output ones digit
                ONES = 1;                // enable ones display
                while (TMR0_2)          // wait for TMR0<2> to go low
                ;
            }
        }
    }
}

/***** FUNCTIONS *****/

/***** Display digit on 7-segment display *****/
void set7seg(uint8_t digit)
{
    // pattern table for 7 segment display on port B
    const uint8_t pat7segB[10] = {
        // RB4 = E, RB1:0 = FG
        0b010010, // 0
        0b000000, // 1
        0b010001, // 2
        0b000001, // 3
        0b000011, // 4
        0b000011, // 5
        0b010011, // 6
        0b000000, // 7
        0b010011, // 8
        0b000011 // 9
    };
};

```

```

// pattern table for 7 segment display on port C
const uint8_t pat7segC[10] = {
    // RC4:1 = CDBA
    0b011110, // 0
    0b010100, // 1
    0b001110, // 2
    0b011110, // 3
    0b010100, // 4
    0b011010, // 5
    0b011010, // 6
    0b010110, // 7
    0b011110, // 8
    0b011110 // 9
};

// Disable displays
PORTB = 0; // clear all digit enable lines on PORTB
PORTC = 0; // and PORTC

// Output digit pattern
PORTB = pat7segB[digit]; // lookup and output port B and C patterns
PORTC = pat7segC[digit];
}

```

CCS PCB

Converting this program for the CCS compiler isn't difficult; it supports the same program structures, such as functions, as the XC8 compiler, and no new features are needed.

Using the `get_timer0()` function, the macro for accessing `TMR0<2>` would be written as:

```
#define TMR0_2 (get_timer0() & 1<<2) // access to TMR0<2>
```

Alternatively, as we saw in [lesson 3](#), `TMR0<2>` could be accessed through a bit variable, declared as:

```
#bit TMR0_2 = 0x01.2 // access to TMR0<2>
```

The main problem with this approach is that it's not portable – you shouldn't assume that `TMR0` will always be at address `01h`; if you migrate your code to another PIC, you may have to remember to change this line. On the other hand, the `get_timer0()` function will always work.

Complete program

Most of the code is very similar to the XC8 version, with register accesses replaced with their CCS built-in function equivalents:

```

/*****
 *
 * Description: Lesson 5, example 2
 *
 * Demonstrates use of multiplexing to drive multiple 7-seg displays
 *
 * 3-digit 7-segment LED display: 1 digit minutes, 2 digit seconds
 * counts in seconds 0:00 to 9:59 then repeats,
 * with timing derived from int 4 MHz oscillator
 *
 *****/
 *
 * Pin assignments:
 * RB0-1, RB4, RC1-4 = 7-segment display bus (common cathode)
 *

```

```

*          RC5          = minutes enable (active high)          *
*          RB5          = tens enable                          *
*          RC0          = ones enable                          *
*                                                                *
*****/

#include <16F506.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,IOSC4,INTRC_IO

// Pin assignments
#define MINUTES PIN_C5          // minutes enable
#define TENS     PIN_B5          // tens enable
#define ONES     PIN_C0          // ones enable

/***** PROTOTYPES *****/
void set7seg(unsigned int8 digit);    // display digit on 7-segment display

/***** MACROS *****/
#define TMR0_2 (get_timer0() & 1<<2) // access to TMR0<2>

/***** MAIN PROGRAM *****/
void main()
{
    unsigned int8    mpx_cnt;        // multiplex counter
    unsigned int8    mins, secs;     // time counters

    //*** Initialisation

    // configure ports
    setup_adc_ports(NO_ANALOGS);     // disable all analog and comparator inputs
    setup_comparator(NC_NC_NC_NC);   // -> RB0, RB1, RC0, RC1 digital

    // configure Timer0
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_256); // timer mode, prescale = 256
                                                // -> bit 2 cycles every 2.048 ms

    //*** Main loop
    while (TRUE)
    {
        // count in seconds from 0:00 to 9:59
        for (mins = 0; mins < 10; mins++)
        {
            for (secs = 0; secs < 60; secs++)
            {
                // for each time count, multiplex display for 1 second
                // (display each of 3 digits for 2.048 ms each,
                // so repeat 1000000/2048/3 times to make 1 second)
                for (mpx_cnt = 0; mpx_cnt < 1000000/2048/3; mpx_cnt++)
                {
                    // display minutes for 2.048 ms
                    while (!TMR0_2) // wait for TMR0<2> to go high
                        ;
                    set7seg(mins); // output minutes digit
                }
            }
        }
    }
}

```

```

        output_high(MINUTES);    // enable minutes display
        while (TMR0_2)          // wait for TMR0<2> to go low
            ;

        // display tens for 2.048 ms
        while (!TMR0_2)        // wait for TMR0<2> to go high
            ;
        set7seg(secs/10);       // output tens digit
        output_high(TENS);      // enable tens display
        while (TMR0_2)         // wait for TMR0<2> to go low
            ;

        // display ones for 2.048 ms
        while (!TMR0_2)        // wait for TMR0<2> to go high
            ;
        set7seg(secs%10);       // output ones digit
        output_high(ONES);      // enable ones display
        while (TMR0_2)         // wait for TMR0<2> to go low
            ;
    }
}
}
}
}

```

```

/***** FUNCTIONS *****/

```

```

/***** Display digit on 7-segment display *****/

```

```

void set7seg(unsigned int8 digit)
{
    // pattern table for 7 segment display on port B
    const int8 pat7segB[10] = {
        // RB4 = E, RB1:0 = FG
        0b010010, // 0
        0b000000, // 1
        0b010001, // 2
        0b000001, // 3
        0b000011, // 4
        0b000011, // 5
        0b010011, // 6
        0b000000, // 7
        0b010011, // 8
        0b000011 // 9
    };

    // pattern table for 7 segment display on port C
    const int8 pat7segC[10] = {
        // RC4:1 = CDBA
        0b011110, // 0
        0b010100, // 1
        0b001110, // 2
        0b011110, // 3
        0b010100, // 4
        0b011010, // 5
        0b011010, // 6
        0b010110, // 7
        0b011110, // 8
        0b011110 // 9
    };
}

```



```

// Disable displays
output_b(0);           // clear all digit enable lines on PORTB
output_c(0);           // and PORTC

// Output digit pattern
output_b(pat7segB[digit]); // lookup and output port B and C patterns
output_c(pat7segC[digit]);
}

```

Comparisons

Here is the resource usage summary for the 3-digit time count example programs, including the BCD version of the assembler example from [baseline assembler lesson 8](#):

Count_7seg_x3

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	118	104	4
XC8 (Free mode)	60	484	11
CCS PCB	56	180	10

Although the C source code is much shorter than the assembler program, the code generated by the C compilers is much bigger than the hand-written assembler version – the CCS version being nearly twice as large, despite having full optimisation enabled. This is mainly because of the apparently simple division and modulus operations used in the C examples. Something may be very easy to express (leading to shorter source code), but be inefficient to implement – and mathematical operations, even simple integer arithmetic, are a classic example.

And without any optimisation, the XC8 compiler (running in ‘Free mode’) generates very poor code indeed, in this example – nearly five times as big as the assembler version!

Summary

We have seen in this lesson that lookup tables can be effectively implemented in C as initialised arrays qualified as ‘const’. We also saw that C bit-manipulation expressions make it reasonably easy to extract more than one segment display pattern from a single table entry, making it seem natural to use a single lookup table – but that the extra instructions that the compiler generates to perform this pattern extraction may not be worth the savings in lookup table size.

Similarly, we saw that it was quite straightforward to use multiplexing to implement a multi-digit display, without needing to be as concerned (as we were with assembly language) about how to store the values being displayed. This allowed us to use simple arithmetic expressions such as ‘secs/10’ and as ‘secs%10’, but at a significant cost in generated code size – demonstrating that what seems easy or natural in C, is not always the most efficient way to do something.

Although it would be possible to re-write the C programs so that the compilers can generate more efficient code, to some extent that misses the point of programming in C – it’s all about being able to save valuable time.

Of course it is useful, when using C, to be aware of which program structures use more memory or need more instructions to implement than others (such as including floating point calculations when it is not necessary). But if you really need efficiency, as you often will with these small devices, it’s difficult to do beat assembler.

The [next lesson](#) ventures into analog territory, covering comparators and programmable voltage references (revisiting material from [baseline assembler lesson 9](#)).

Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 6: Analog Comparators

[Baseline assembler lesson 9](#) explained how to use the analog comparators and absolute and programmable voltage references available on baseline PICs, such as the PIC16F506, using assembly language. This lesson demonstrates how to use C to access those facilities, re-implementing the examples using Microchip's XC8 (running in "Free mode") and CCS' PCB compilers¹.

In summary, this lesson covers:

- Basic use of the analog comparator modules available on the PIC16F506
- Using the internal absolute 0.6 V voltage reference
- Configuring and using the internal programmable voltage reference
- Enabling comparator output, to facilitate the addition of external hysteresis
- Wake-up on comparator change
- Driving Timer0 from a comparator output

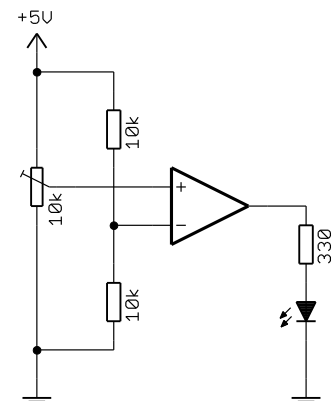
with examples for XC8 and CCS PCB.

Comparators

As we saw in [baseline assembler lesson 9](#), an *analog comparator* is a device which compares the voltages present on its positive and negative inputs. In normal (non-inverted) operation, the comparator's output is set to a logical "high" only when the voltage on the positive input is greater than that on the negative input; otherwise the output is "low". As such, they provide an interface between analog and digital circuitry.

In the circuit shown on the right, the comparator output will go high, lighting the LED, only when the potentiometer is set to a position past "half-way", i.e. positive input is greater than 2.5 V.

Comparators are typically used to detect when an analog input is above or below some threshold (or, if two comparators are used, within a defined band) – very useful for working with many types of real-world sensors. They are also used with digital inputs to match different logic levels, and to shape poorly defined signals.



¹ XC8 is available as a free download from www.microchip.com, and CCS PCB is bundled for free with MPLAB 8

Comparator 1

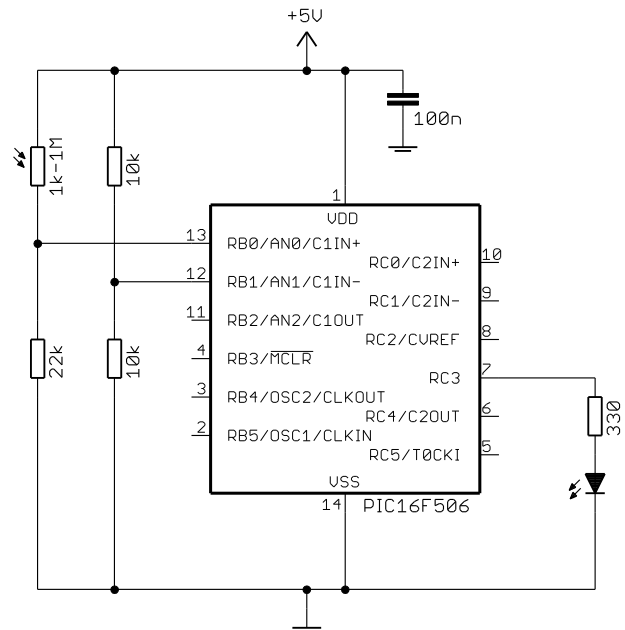
In [baseline assembler lesson 9](#), the circuit on the right, which includes a light dependent resistor (LDR, or CdS photocell), was used to demonstrate the basic operation of Comparator 1, the simpler of the two comparator modules in the PIC16F506.

The exact resistance range of the photocell is not important, but would ideally have a resistance of around 20 k Ω or so for normal indoor lighting conditions, so that the voltage at C1IN+ will vary around 2.5 V or so.

If you have the [Gooligum baseline training board](#), you can implement this circuit by placing a shunt across pins 2 and 3 ('LDR1') of JP24, connecting the photocell in the lower left of the board (PH1) to C1IN+, and in JP19 to enable the LED on RC3.

The connection to C1IN- (labelled 'GP/RA/RB1' on the board) is available as pin 9 on the 16-pin header. +V and GND are brought out on pins 15 and 16, respectively, making it easy to add the 10 k Ω resistors (supplied with the board), forming a voltage divider, by using the solderless breadboard.

If you are using the Microchip Low Pin Count Demo Board, the 10 k Ω potentiometer on that board, and the 1 k Ω resistor in series between it and C1IN+, must be used as the "fixed" resistance, forming the lower arm of the potential divider. You must also remove jumper JP5 (you may need to cut the PCB trace – ideally you'd install a jumper, so that you can reconnect it again later), to disconnect the pot from the +5 V supply. If you turn the pot all the way to the right, you'll have a total resistance of 11 k Ω between C1IN+ and ground. That means that ideally you'd use a photocell with a resistance of around 10 k Ω with normal indoor lighting. The photocell can then be connected between pin 7 on the 14-pin header and +5 V. Note that if you do not have a photocell available, you can still explore comparator operation by connecting the C1IN+ input directly to the centre tap on the 10 k Ω potentiometer.



We saw in [baseline assembler lesson 9](#) that, to configure Comparator 1 to behave like the standalone comparator shown on the previous page, where the output bit (C1OUT) indicates that the voltage on the C1IN+ input is higher than that on the C1IN- input, it is necessary to set the C1PREF, C1NREF and C1POL bits in the CM1CON0 register, and to turn on the comparator module by setting the C1ON bit:

```
movlw    1<<C1POL|1<<C1ON|1<<C1PREF|1<<C1NREF
;        ; pos ref is C1IN+ (C1PREF = 1)
;        ; neg ref is C1IN- (C1NREF = 1)
;        ; normal polarity (C1POL = 1)
;        ; comparator on (C1ON = 1)
movwf   CM1CON0
;        ; -> C1OUT = 1 if C1IN+ > C1IN-
```

The LED attached to RC3 was turned on when the comparator output was high (C1OUT = 1) by:

```
loop    btfsc   CM1CON0,C1OUT    ; if comparator output high
        bsf     LED              ; turn on LED
        btfss  CM1CON0,C1OUT    ; if comparator output low
        bcf     LED              ; turn off LED

        goto   loop              ; repeat forever
```

XC8

We saw in [lesson 3](#) that symbols, defined in the XC8 header files, can be used to represent register bits to construct a value to load into the `OPTION` register, for example:

```
OPTION = ~T0CS & ~PSA | 0b1111;
```

However the `OPTION` register is an exception. The XC8 header files define most special function registers, including `CM1CON0`, as unions of structures containing bit-fields corresponding to that register's bits.

If you wish to set and/or clear a number of bits in a register such as `CM1CON0` at once, you can use a numeric constant such as:

```
CM1CON0 = 0b00101110;           // configure comparator 1:
//--1-----                    normal polarity (C1POL = 1)
//----1---                      comparator on (C1ON = 1)
//-----1--                    -ref is C1IN- (C1NREF = 1)
//-----1-                     +ref is C1IN+ (C1PREF = 1)
//                               -> C1OUT = 1 if C1IN+ > C1IN-
```

This is ok, as long as you express the value in binary, so that it is obvious which bits are being set or cleared, and clearly commented, as above.

It is certainly much clearer than the equivalent:

```
CM1CON0 = 46;
```

However, a more natural way to approach this, in XC8, is to use a sequence of assignments, to set or clear the appropriate register bits via the bit-fields defined in the header files.

For example:

```
// configure comparator 1
CM1CON0bits.C1PREF = 1;         // +ref is C1IN+
CM1CON0bits.C1NREF = 1;         // -ref is C1IN-
CM1CON0bits.C1POL = 1;          // normal polarity (C1IN+ > C1IN-)
CM1CON0bits.C1ON = 1;           // turn comparator on
```

This is clear and easy to maintain, but a series of single-bit assignments like this requires more program memory than a whole-register assignment. It is also no longer an *atomic* operation, where all the bits are updated at once. This can be an important consideration in some instances², but it is not relevant here. Note also that the remaining bits in `CM1CON0` are not being explicitly set or cleared; that is ok because in this example we don't care what values they have.

The comparator's output bit, `C1OUT`, is available as the single-bit bit-field '`CM1CON0bits.C1OUT`'.

For example:

```
LED = CM1CON0bits.C1OUT;        // turn on LED iff comparator output high
```

With the above configuration, the LED will turn on when the LDR is illuminated.

If instead you wanted it to operate the other way, so that the LED is lit when the LDR is in darkness, you could invert the comparator output test, so that the LED is set high when `C1OUT` is low:

```
LED = !CM1CON0bits.C1OUT;       // turn on LED iff comparator output low
```

² this can be a consideration with mid-range PICs, where interrupts may be used

Alternatively, you can configure the comparator so that its output is inverted, using either:

```
CM1CON0 = 0b00001110;    // configure comparator 1:
    //--0-----          inverted polarity (C1POL = 0)
    //----1----          comparator on (C1ON = 1)
    //-----1--          -ref is C1IN- (C1NREF = 1)
    //-----1-          +ref is C1IN+ (C1PREF = 1)
    //                   -> C1OUT = 1 if C1IN+ < C1IN-
```

or:

```
// configure comparator 1
CM1CON0bits.C1PREF = 1;    // +ref is C1IN+
CM1CON0bits.C1NREF = 1;    // -ref is C1IN-
CM1CON0bits.C1POL = 0;    // inverted polarity (C1IN+ < C1IN-)
CM1CON0bits.C1ON = 1;     // turn comparator on
```

Complete program

Here is the complete inverted polarity version of the program, for XC8:

```
*****
*   Description:    Lesson 6, example 1b                               *
*                                                         *
*   Demonstrates basic use of Comparator 1 polarity bit           *
*                                                         *
*   Turns on LED when voltage on C1IN+ < voltage on C1IN-       *
*                                                         *
*****
*   Pin assignments:                                             *
*   C1IN+ = voltage to be measured (e.g. pot output or LDR)      *
*   C1IN- = threshold voltage (set by voltage divider resistors) *
*   RC3   = indicator LED                                         *
*                                                         *
*****/

#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & IOSCFSS_OFF & OSC_IntRC_RB4EN);

// Pin assignments
#define LED    PORTCbits.RC3    // indicator LED on RC3
#define nLED   3                // (port bit 3)

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure ports
    TRISC = ~(1<<nLED);          // configure LED pin (only) as an output

    // configure Comparator 1
    CM1CON0bits.C1PREF = 1;      // +ref is C1IN+
    CM1CON0bits.C1NREF = 1;      // -ref is C1IN-
    CM1CON0bits.C1POL = 0;      // inverted polarity (C1IN+ < C1IN-)
    CM1CON0bits.C1ON = 1;       // turn comparator on
```

```

    /*** Main loop
    for (;;)
    {
        // continually display comparator output
        LED = CM1CON0bits.C1OUT;
    }
}

```

CCS PCB

As we've come to expect, the CCS PCB compiler provides a built-in function for configuring the comparators: 'setup_comparator()'.

It is used with symbols defined in the device-specific header files. For example, "16F506.h" contains:

```

//Pick one constant for COMP1
#define CP1_B0_B1      0x3000000E
#define CP1_B0_VREF   0x1000000A
#define CP1_B1_VREF   0x20000008

//Optionally OR with one or both of the following
#define CP1_OUT_ON_B2 0x04000040
#define CP1_INVERT    0x00000020
#define CP1_WAKEUP    0x00000001
#define CP1_TIMER0    0x00000010

```

The first set of three symbols defines the positive and negative inputs for the comparator; one of these must be specified. The last set of four symbols are used to select comparator options, such as inverted polarity, by ORing them into the expression passed to the 'setup_comparator()' function.

For example, to use C1IN+ (which shares its pin with RB0) as the positive input, and C1IN- (which shares its pin with RB1) as the negative input, with normal polarity:

```
setup_comparator(CP1_B0_B1); // C1 on, C1OUT = 1 if C1IN+ > C1IN-
```

To turn off the comparator, use:

```
setup_comparator(NC_NC_NC_NC); // turn off comparators 1 and 2
```

This actually turns off both comparator modules on the PIC16F506. If setup_comparator() is used to configure only one of the comparators, the other is turned off. We'll see later how to configure both comparators.

To make it clear that comparator 2 is being turned off, when setting up comparator 1, you can write:

```
setup_comparator(CP1_B0_B1|NC_NC); // C1 on, C1OUT = 1 if C1IN+ > C1IN-
// (disable C2)
```

Like XC8, CCS PCB makes the C1OUT bit available as the single-bit variable 'C1OUT', so to copy the comparator output to the LED, we can use:

```
output_bit(LED,C1OUT); // turn on LED iff comparator output high
```

To invert the operation of this circuit, so that the LED turns on when the LDR is in darkness, you could copy the inverse of the comparator output to the LED, using:

```
output_bit(LED,~C1OUT); // turn on LED iff comparator output low
```

Alternatively, you could configure the comparator for inverted output, using:

```
setup_comparator(CP1_B0_B1|CP1_INVERT); // C1 on, C1OUT = 1 if C1IN+ < C1IN-
```

Complete program

Here is the complete inverted polarity version of the program, for CCS PCB:

```
*****
*
* Description: Lesson 6, example 1b
*
* Demonstrates basic use of Comparator 1 polarity bit
*
* Turns on LED when voltage on C1IN+ < voltage on C1IN-
*
*****
*
* Pin assignments:
* C1IN+ = voltage to be measured (e.g. pot output or LDR)
* C1IN- = threshold voltage (set by voltage divider resistors)
* RC3 = indicator LED
*
*****/

#include <16F506.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,IOSC4,INTRC_IO

// Pin assignments
#define LED PIN_C3 // indicator LED on RC3

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure comparators
    setup_comparator(CP1_B0_B1|CP1_INVERT); // C1 on, C1OUT = 1 if C1IN+ < C1IN-

    /*** Main loop
    while (TRUE)
    {
        // continually display comparator output
        output_bit(LED,C1OUT);
    }
}
```


Comparisons

The following table summarises the resource usage for the “comparator 1 inverted polarity” assembler and C example programs.

Comp1_LED-neg

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	20	14	0
XC8 (Free mode)	12	29	0
CCS PCB	7	28	4

The CCS version is amazingly short, at only 7 lines of code – demonstrating again that the use of built-in functions, for actions such as configuring comparators, can lead to very compact code. The XC8 version is longer, because separate statements are used to set or clear each bit in `CM1CON0` – but still only around half as long as the assembler version.

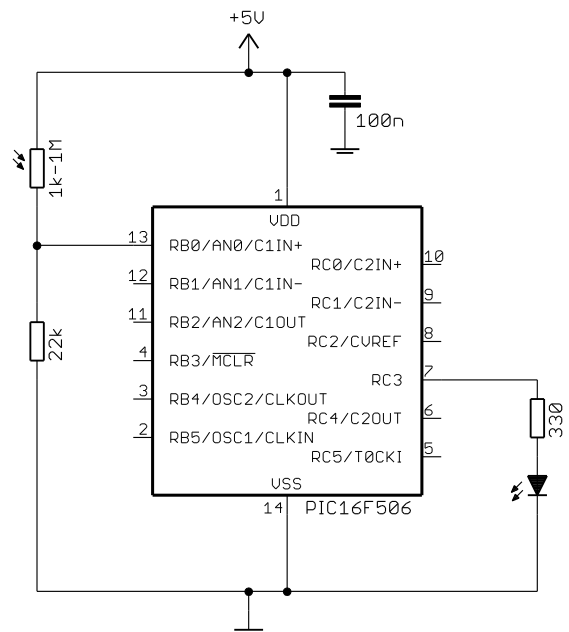
Absolute Voltage Reference

It is possible to assign an internal 0.6 V reference as the negative input for comparator 1.

This means that the external 10 kΩ resistors, forming a voltage divider in the previous example, are unnecessary, and they can be removed – as in the circuit on the right. You should find that removing the resistors makes no difference to the circuit’s operation.

It also means that the RB1 pin is now available for use.

To select the internal 0.6 V reference as the internal input, clear the `C1NREF` bit in the `CM1CON0` register.



XC8

Assuming that we still want inverted operation, where the LED is on when the LDR is in darkness, simply change the comparator configuration instructions to:

```
// configure comparator 1
CM1CON0bits.C1PREF = 1; // +ref is C1IN+
CM1CON0bits.C1NREF = 0; // -ref is 0.6 V internal ref
CM1CON0bits.C1POL = 0; // inverted polarity (C1IN+ < 0.6 V)
CM1CON0bits.C1ON = 1; // turn comparator on
```

Or alternatively:

```
CM1CON0 = 0b00001010;    // configure comparator 1:
//--0-----             // inverted polarity (C1POL = 0)
//----1----             // comparator on (C1ON = 1)
//-----0--             // -ref is 0.6 V (C1NREF = 0)
//-----1-             // +ref is C1IN+ (C1PREF = 1)
//                      // -> C1OUT = 1 if C1IN+ < 0.6 V
```

CCS PCB

To use the internal 0.6 V reference, we only need to change the parameter in the `setup_comparator()` function:

```
setup_comparator(CP1_B0_VREF|CP1_INVERT); // C1 on: C1IN+ < 0.6 V
```

This specifies that RB0 (C1IN+) be used as the positive input and the 0.6 V reference be used as the negative input on comparator 1, with the output inverted.

External Output and Hysteresis

As was explained in [baseline assembler lesson 9](#), it is often desirable to add hysteresis to a comparator, to make it less sensitive to small changes in the input signal due to superimposed noise or other interference. For example, in the above examples using a photocell, you will find that the output LED flickers when the light level is close to the threshold, particularly with mains-powered artificial illumination, which varies at 50 or 60 Hz.

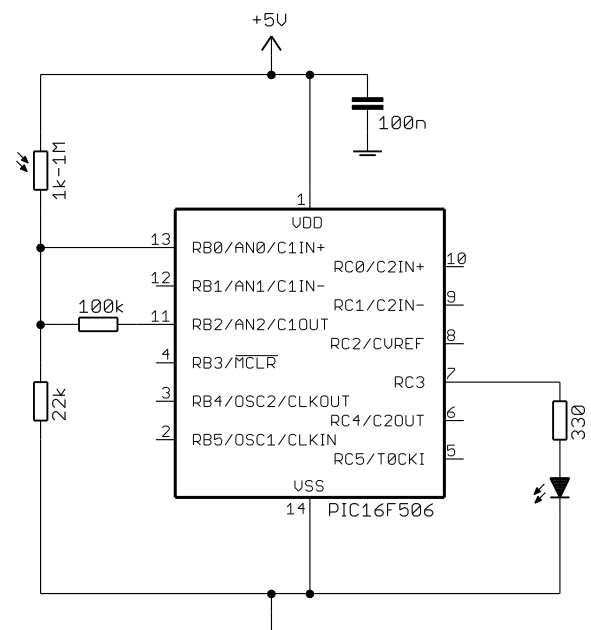
On the PIC16F506, the output of comparator 1 can be made available on the C1OUT pin.

This is done by clearing the `C1OUTEN` bit in `CM1CON0`.

In the circuit on the right, hysteresis has been introduced by using a 100 kΩ resistor to feed some of the comparator's output, on C1OUT, back into the C1IN+ input.

You can build this with the [Gooligum baseline training board](#) by placing the supplied 100 kΩ resistor between pins 8 ('GP/RA/RB0') and 13 ('GP/RA/RB2') on the 16-pin header.

Or, if you are using the Microchip Low Pin Count Demo Board, you would place the feedback resistor between pins 7 and 9 on the 14-pin header.



Note that, because C1OUT shares its pin with the AN2 analog input, the comparator output will not appear on C1OUT until the AN2 input is disabled; as we'll see in the [next lesson](#), a simple way to disable all the analog inputs is to clear the `ADCON0` register³.

Note also that, because hysteresis relies on positive feedback, the comparator output must not be inverted.

³ See [baseline assembler lesson 10](#) for an explanation of the `ADCON0` register.

XC8

To clear `ADCON0` (disabling all analog inputs, including `AN2`, to make it possible for the output of comparator 1 to be placed on the `C1OUT` pin) using `XC8`, we can write:

```
ADCON0 = 0; // disable analog inputs -> C1OUT usable
```

Comparator 1 can then be configured by:

```
// configure comparator 1
CM1CON0bits.C1PREF = 1; // +ref is C1IN+
CM1CON0bits.C1NREF = 0; // -ref is 0.6 V internal ref
CM1CON0bits.C1POL = 1; // normal polarity (C1IN+ > 0.6 V)
CM1CON0bits.nC1OUTEN = 0; // enable C1OUT (for hysteresis feedback)
CM1CON0bits.C1ON = 1; // turn comparator on
```

or:

```
CM1CON0 = 0b00101010; // configure comparator 1:
    //-0----- enable C1OUT pin (/C1OUTEN = 0)
    //--1----- normal polarity (C1POL = 1)
    //----1--- comparator on (C1ON = 1)
    //-----0-- -ref is 0.6 V (C1NREF = 0)
    //-----1- +ref is C1IN+ (C1PREF = 1)
    //          -> C1OUT = 1 if C1IN+ > 0.6V,
    //          C1OUT enabled (for hysteresis feedback)
```

Note that the external output is enabled by clearing `C1OUTEN` and that the output is not inverted.

Since we want to light the LED when the photocell is in darkness (`C1IN+ < 0.6 V`), we need to invert the display logic:

```
LED = !CM1CON0bits.C1OUT; // display comparator output (inverted)
```

CCS PCB

As we'll see in the [next lesson](#), the CCS compiler provides a `setup_adc_ports()` built-in function, which is used to select, among other things, whether pins are analog or digital.

It can be used to disable all the analog inputs, as follows:

```
setup_adc_ports(NO_ANALOGS); // disable analog inputs -> C1OUT usable
```

To enable `C1OUT` (which shares its pin with `RB2`), OR the `CP1_OUT_ON_B2` symbol into the parameter passed to the `setup_comparator()` function:

```
setup_comparator(CP1_B0_VREF|CP1_OUT_ON_B2); // C1 on: C1IN+ > 0.6 V
// C1OUT enabled
```

Note again that, to make hysteresis possible, the comparator output bit is no longer inverted.

If we still want the LED to indicate darkness (`C1IN+ < 0.6 V`), we have to invert the display logic instead:

```
// display comparator output (inverted)
output_bit(LED, ~C1OUT);
```

Wake-up on Comparator Change

We saw in [baseline assembler lesson 9](#) that the comparator modules in the PIC16F506 can be used to wake the device from sleep when the comparator output changes – useful for conserving power while waiting for a signal change from a sensor.

To enable wake-up on change for Comparator 1, clear the $\overline{C1WU}$ bit in the CM1CON0 register.

To determine whether a reset was due to a wake-up on comparator change, test the CWUF flag in the STATUS register. If CWUF is set, we can be sure that the device has been woken from sleep by a comparator change. If it is clear, some other type of reset has occurred.

Note that there is no indication of which comparator was the source of the reset. If you have configured both comparators for wake-up on change, you need to store the previous values of their outputs, so that you can determine which one changed.

In the example in [baseline assembler lesson 9](#), the previous circuit was used (keeping the hysteresis, making the comparator less sensitive), with the LED indicating when a comparator change occurs, by lighting for one second. While waiting for a comparator change, the PIC was placed into sleep mode – immediately after reading CM1CON0 to prevent false triggering.

XC8

The CWUF flag can be tested directly, so we can simply write:

```
if (!STATUSbits.CWUF)
{
    // power-on reset
}
else
{
    // wake-up on comparator change occurred
}
```

The test is inverted here so that the normal power-on initialisation code appears first – it seems clearer that way, since you would normally look toward the start of a program to find the initialisation code.

The comparator configuration code is similar to what we've seen before, with the addition of "CM1CON0bits.nC1WU = 0;" to enable the wake-up on change function:

```
// configure comparator 1
CM1CON0bits.C1PREF = 1;    // +ref is C1IN+
CM1CON0bits.C1NREF = 0;    // -ref is 0.6 V internal ref
CM1CON0bits.C1POL = 1;    // normal polarity (C1IN+ > 0.6 V)
CM1CON0bits.nC1OUTEN = 0; // enable C1OUT (for hysteresis feedback)
CM1CON0bits.nC1WU = 0;   // enable wake-up on change
CM1CON0bits.C1ON = 1;     // turn comparator on
```

Or, this could have been written as:

```
CM1CON0 = 0b00101010;    // configure comparator 1:
    // -0-----    enable C1OUT pin (/C1OUTEN = 0)
    // --1-----    normal polarity (C1POL = 1)
    // ----1---    comparator on (C1ON = 1)
    // -----0--    -ref is 0.6 V (C1NREF = 0)
    // -----1-    +ref is C1IN+ (C1PREF = 1)
    // -----0    enable wake on change (/C1WU = 0)
```

The comparator initialisation is followed by a delay of 10 ms, allowing the comparator to settle before the device is placed into sleep mode, to avoid initial false triggering.

As another (necessary) precaution to avoid false triggering, we read the current value of `CM1CON0`, immediately before entering sleep mode:

```
CM1CON0;           // read comparator to clear mismatch condition
SLEEP();           // enter sleep mode
```

Any statement which reads `CM1CON0` could be used.

“`CM1CON0`” is an expression which evaluates to the value of the contents of `CM1CON0`, but does nothing.

In general, the compiler’s optimiser will discard any such “do nothing” statements.

However, `CM1CON0` is declared as a ‘volatile’ variable in the processor header file. This qualifier tells the compiler that the value of this variable may change at any time, to prevent the optimiser from eliminating apparently redundant references to it. It also ensures that, when the variable’s name is used on its own in this way, the compiler will generate code which reads the variable’s memory location and discards the result, which is exactly what we want.

Complete program

Here is how the above code fragments fit together, within the complete “wake-up on comparator change demo” program:

```

/*****
*
* Description: Lesson 6, example 2
*
* Demonstrates wake-up on comparator change
*
* Turns on LED for 1s when comparator 1 output changes,
* then sleeps until the next change
* (internal 0.6 V reference with hysteresis)
*
*****/
*
* Pin assignments:
*   CLIN+ = voltage to be measured (e.g. pot output or LDR)
*   C1OUT = comparator output (fed back to input via resistor)
*   RC3   = indicator LED
*
*****/

#include <xc.h>

#define _XTAL_FREQ 4000000 // oscillator frequency for delay functions

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & IOSCFE_OFF & OSC_IntRC_RB4EN);

// Pin assignments
#define LED PORTCbits.RC3 // indicator LED on RC3
#define nLED 3 // (port bit 3)

/***** MAIN PROGRAM *****/
void main()
{

```

```

//*** Initialisation

// configure ports
LED = 0; // start with LED off
TRISC = ~(1<<nLED); // configure LED pin (only) as an output
ADCON0 = 0; // disable analog inputs -> C1OUT usable

// check for wake-up on comparator change
if (!STATUSbits.CWUF)
{
    // power-on reset has occurred:

    // configure comparator 1
    CM1CON0bits.C1PREF = 1; // +ref is C1IN+
    CM1CON0bits.C1NREF = 0; // -ref is 0.6 V internal ref
    CM1CON0bits.C1POL = 1; // normal polarity (C1IN+ > 0.6 V)
    CM1CON0bits.nC1OUTEN = 0; // enable C1OUT (for hysteresis feedback)
    CM1CON0bits.nC1WU = 0; // enable wake-up on change
    CM1CON0bits.C1ON = 1; // turn comparator on

    // delay 10 ms to allow comparator to settle
    __delay_ms(10);
}
else
{
    // wake-up on comparator change occurred:

    // flash LED
    LED = 1; // turn on LED
    __delay_ms(1000); // delay 1 sec
}

//*** Sleep until comparator change
LED = 0; // turn off LED
CM1CON0; // read comparator to clear mismatch condition
SLEEP(); // enter sleep mode
}

```

CCS PCB

In [lesson 4](#), we saw that, although CCS PCB provides a 'restart_cause()' function, which returns a value indicating why the device has been reset, it does not support wake on pin change resets.

Unfortunately, this function does not support wake on comparator change resets either.

Instead, we need to test the CWUF flag, which the PCB compiler does not normally provide direct access to. The solution, as we saw in [lesson 4](#), is to use the #bit directive, as follows:

```
#bit CWUF = 0x03.6 // CWUF flag in STATUS register
```

This flag can then be referenced directly, in the same way as we did with XC8:

```

if (!CWUF)
{
    // power-on reset
}
else
{
    // wake-up on comparator change occurred
}

```

To configure comparator 1 for wake-up on change, OR the 'CP1_WAKEUP' symbol into the parameter passed to the `setup_comparator()` function:

```
setup_comparator(CP1_B0_VREF|CP1_OUT_ON_B2|CP1_WAKEUP);
```

For clarity, you may wish to split this function call across multiple lines, so that the symbols can be commented separately:

```
setup_comparator(CP1_B0_VREF|          // C1 on: C1IN+ > 0.6V,
                 CP1_OUT_ON_B2|       //          C1OUT pin enabled,
                 CP1_WAKEUP);         //          wake-up on change enabled
```

We cannot use the same method as we did with XC8 to read the current comparator output prior to entering sleep mode, because the CCS PCB compiler will not generate any instructions when an expression is not used for anything. So, to read a bit or register, we must assign it to a variable.

Since the PCB compiler exposes the C1OUT bit, we can use:

```
temp = C1OUT;          // read comparator to clear mismatch condition
sleep();              // enter sleep mode
```

Since C1OUT is a single-bit variable, the `temp` variable can be declared to be 'int1' (single bit), although 'int8' (one byte, or 8 bits) is also appropriate; the generated code size is the same for both.

Complete program

The following listing shows how these code fragments fit into the CCS PCB version of the “wake-up on comparator change demo” program:

```

/*****
 *
 * Description: Lesson 6, example 2
 *
 * Demonstrates wake-up on comparator change
 *
 * Turns on LED for 1 s when comparator 1 output changes,
 * then sleeps until the next change
 * (internal 0.6 V reference with hysteresis)
 *
 *****/
 *
 * Pin assignments:
 * C1IN+ = voltage to be measured (e.g. pot output or LDR)
 * C1OUT = comparator output (fed back to input via resistor)
 * RC3 - indicator LED
 *
 *****/
#include <16F506.h>

#define CWUF 0x03.6 // CWUF flag in STATUS register

#define OSC_FREQ 4000000 // oscillator frequency for delay_ms()

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
#define MCLR, NOPROTECT, NOWDT, INTRC_IO, IOSC4

// Pin assignments
#define LED PIN_C3 // indicator LED on RC3

```

```

/***** MAIN PROGRAM *****/
void main()
{
    int1    temp;                // temp variable for reading C1

    //*** Initialisation

    // configure ports
    setup_adc_ports(NO_ANALOGS);    // disable analog inputs -> C1OUT usable

    // check for wake-up on comparator change
    if (!CWUF)
    {
        // power-on reset has occurred:

        // configure comparators
        setup_comparator(CP1_B0_VREF|    // C1 on: C1IN+ > 0.6V,
                        CP1_OUT_ON_B2|  // C1OUT pin enabled,
                        CP1_WAKEUP);    // wake-up on change enabled

        // delay 10 ms to allow comparator to settle
        delay_ms(10);
    }
    else
    {
        // wake-up on comparator change occurred:

        // flash LED

        output_high(LED);            // turn on LED
        delay_ms(1000);              // delay 1 sec
    }

    //*** Sleep until comparator change
    output_low(LED);                // turn off LED
    temp = C1OUT;                   // read comparator to clear mismatch condition
    sleep();                         // enter sleep mode
}

```

Comparisons

Here is the resource usage for the “wake-up on comparator change demo” assembler and C examples.

Comp1_Wakeup

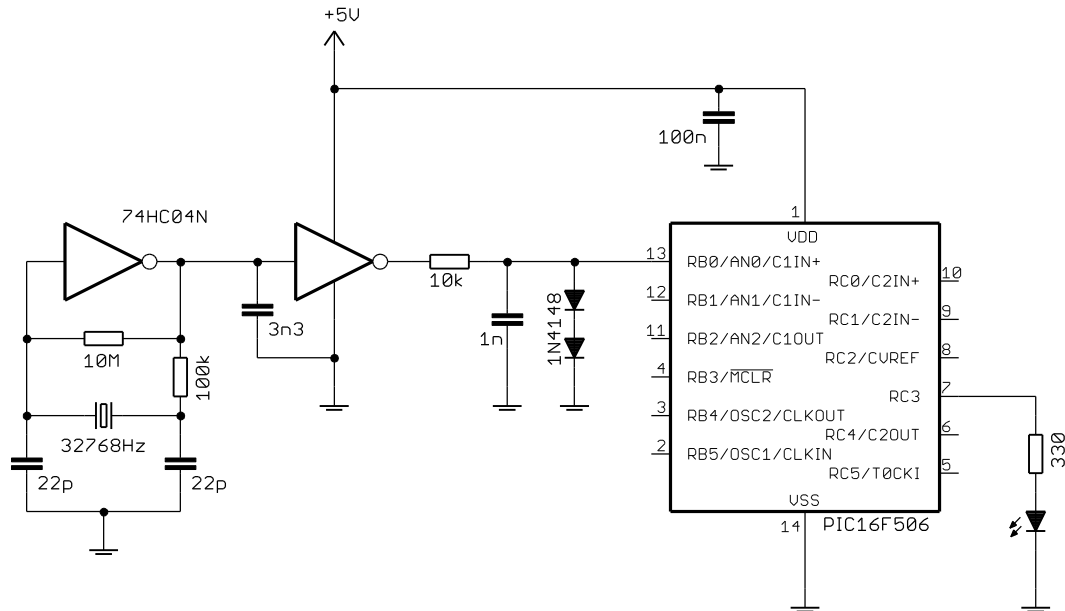
Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	33	43	3
XC8 (Free mode)	23	58	3
CCS PCB	17	66	7

Although the CCS source code continues to be the shortest, the CCS compiler generates the least efficient code in this example – even larger than the (unoptimised) XC8 version.

Clocking Timer0

As explained in [baseline assembler lesson 9](#), the output of comparator 1 can be used to clock Timer0. This is useful for counting pulses which do not meet the signal requirements for digital inputs (specified in the “DC Characteristics” table in the device data sheet).

To demonstrate this, the circuit below was used. The external clock module from [baseline assembler lesson 5](#) is used to supply a “clean” 32.768 kHz signal, which is degraded by being passed through a low-pass filter and clipped by two diodes, creating a signal with 32.768 kHz pulses which peak at around 1 V.



This circuit can be built with the [Gooligum baseline training board](#) and the supplied 10 k Ω resistor, 1 nF capacitor and two 1N4148 diodes. The 32.768 kHz oscillator output is available on pin 1 (‘32 kHz’) of the 16-pin header, the C1IN+ input is pin 8 (‘GP/RA/RB0’) and ground is pin 16 (‘GND’). You should also remove the shunt from JP24 (disconnecting the pot or photocell from C1IN+).

Note: you must only connect these additional components to C1IN+ **after** programming the PIC, to avoid interference with the programming process. You need to program the PIC before making the connection to C1IN+. You can then apply power (whether from a PICKit 2, PICKit3, or external power supply) and release reset – and the LED on RC3 should start flashing.

The degraded signal cannot be used to drive a digital input directly, but the clock pulses can be detected by a comparator with a 0.6 V input voltage reference.

The example program in [baseline assembler lesson 9](#) used Timer0, driven from the 32.768 kHz clock, via comparator 1, to flash the LED at 1 Hz. This was done by assigning the prescaler to Timer0, selecting a prescale ratio of 1:128, and then copying the value to TMR0<7> (which is then cycling at 1 Hz) to the LED output.

To use comparator 1 as the source for Timer0, clear the $\overline{\text{C1T0CS}}$ bit in the CM1CON0 register (to enable the comparator 1 timer output), and set the T0CS bit in the OPTION register (to select Timer0 external counter mode).

XC8

As we saw in [lesson 3](#), to configure Timer0 for counter mode, using XC8, we can use:

```
OPTION = T0CS & ~PSA | 0b110;    // configure Timer0:
                                // counter mode (T0CS = 1)
                                // prescaler assigned to Timer0 (PSA = 0)
                                // prescale = 128 (PS = 110)
                                // -> incr at 256 Hz with 32.768 kHz input
```

To configure comparator 1, with the timer output enabled, we have:

```
// configure comparator 1
CM1CON0bits.C1PREF = 1;        // +ref is C1IN+
CM1CON0bits.C1NREF = 0;        // -ref is 0.6 V internal ref
CM1CON0bits.C1POL = 1;        // normal polarity (C1IN+ > 0.6 V)
CM1CON0bits.nC1T0CS = 0;     // enable TMR0 clock source
CM1CON0bits.C1ON = 1;         // turn comparator on
                                // -> C1OUT = 1 if C1IN+ > 0.6 V,
                                // TMR0 clock from C1
```

We can then copy TMR0<7> to the LED output, using a shadow register, as we've done before:

```
sFLASH = (TMR0 & 1<<7) != 0;    // sFLASH = TMR0<7>

PORTC = sPORTC.port;           // copy shadow to PORTC
```

Complete program

Here is how the code for the “comparator 1 timer output demo” program fits together, using XC8:

```
/******
 *
 * Description:    Lesson 6, example 3
 *
 * Demonstrates use of comparator 1 to clock TMR0
 *
 * LED flashes at 1 Hz (50% duty cycle),
 * with timing derived from 32.768 kHz input on C1IN+
 *
 * *****
 *
 * Pin assignments:
 * C1IN+ = 32.768 kHz signal
 * RC3   = flashing LED
 *
 * *****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & IOSCFs_OFF & OSC_InTRC_RB4EN);

// Pin assignments
#define sFLASH sPORTC.RC3           // flashing LED (shadow)
```

```

/***** GLOBAL VARIABLES *****/
union {
    uint8_t      port;
    struct {
        unsigned RC0      : 1;
        unsigned RC1      : 1;
        unsigned RC2      : 1;
        unsigned RC3      : 1;
        unsigned RC4      : 1;
        unsigned RC5      : 1;
    };
} sPORTC;

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure ports
    TRISC = 0b110111;

    // configure timer
    OPTION = T0CS & ~PSA | 0b110;

    // configure comparator 1
    CM1CON0bits.C1PREF = 1;
    CM1CON0bits.C1NREF = 0;
    CM1CON0bits.C1POL = 1;
    CM1CON0bits.nC1T0CS = 0;
    CM1CON0bits.C1ON = 1;

    /*** Main loop
    for (;;)
    {
        // TMR0<7> cycles at 1 Hz, so continually copy to LED
        sFLASH = (TMR0 & 1<<7) != 0;

        PORTC = sPORTC.port;
    }
}

```

CCS PCB

We saw in [lesson 3](#) that to configure Timer0 for counter mode, using CCS PCB, we can use:

```
setup_timer_0(RTCC_EXT_L_TO_H|RTCC_DIV_128);
```

To enable the timer output on comparator 1, we need to include the 'CP1_TIMER0' symbol in the parameter passed to the `setup_comparator()` function:

```
setup_comparator(CP1_B0_VREF|CP1_TIMER0);           // C1 on: C1IN+ > 0.6 V,
                                                    //          TMR0 clock enabled
```

Finally, bit 7 of TMR0 can be copied to the LED output, using a shadow register, by:

```
sFLASH = (get_timer0() & 1<<7) != 0;           // sFLASH = TMR0<7>
output_c(sPORTC.port);                          // copy shadow to PORTC
```

Complete program

Here is the complete “comparator 1 timer output demo” program, using CCS PCB:

```

/*****
*
* Description: Lesson 6, example 3
*
* Demonstrates use of comparator 1 to clock TMR0
*
* LED flashes at 1 Hz (50% duty cycle),
* with timing derived from 32.768 kHz input on C1IN+
*
*****/
*
* Pin assignments:
* C1IN+ = 32.768 kHz signal
* RC3 = flashing LED
*
*****/
#include <16F506.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC_IO,IOSC4

// Pin assignments
#define sFLASH sPORTC.RC3           // flashing LED (shadow)

/***** GLOBAL VARIABLES *****/
union {
    unsigned int8 port;           // shadow copy of PORTC
    struct {
        unsigned RC0 : 1;
        unsigned RC1 : 1;
        unsigned RC2 : 1;
        unsigned RC3 : 1;
        unsigned RC4 : 1;
        unsigned RC5 : 1;
    };
} sPORTC;

/***** MAIN PROGRAM *****/

```

```

void main()
{
    /*** Initialisation

    // configure Timer0
    setup_timer_0(RTCC_EXT_L_TO_H|RTCC_DIV_128); // counter mode, prescale = 128
                                                // -> increment at 256 Hz
                                                //   with 32.768 kHz input

    // configure comparators
    setup_comparator(CP1_B0_VREF|CP1_TIMER0); // C1 on: C1IN+ > 0.6 V,
                                                //           TMR0 clock enabled

    /*** Main loop
    while (TRUE)
    {
        // TMR0<7> cycles at 1 Hz, so continually copy to LED
        sFLASH = (get_timer0() & 1<<7) != 0; // sFLASH = TMR0<7>

        output_c(sPORTC.port); // copy shadow to PORTC
    }
}

```

Comparator 2 and the Programmable Voltage Reference

As described in greater detail in [baseline assembler lesson 9](#), comparator 2 is very similar to comparator 1, except that:

- A wider range of inputs can be used as the positive reference: C2IN+, C2IN- and C1IN+
- The negative reference can be either the C2IN- pin, or an internal programmable voltage reference
- The fixed 0.6 V internal voltage reference *cannot* be used with comparator 2
- The output of comparator 2 is *not* available as an input to Timer0

Comparator 2 is controlled by the CM2CON0 register.

The programmable voltage reference can be set to one of 32 available voltages, from 0 V to $0.72 \times V_{DD}$.

The reference voltage is set by the VR<3:0> bits and VRR, which selects a high or low voltage range:

VRR = 1 selects the low range, where $CV_{REF} = VR<3:0>/24 \times V_{DD}$.

VRR = 0 selects the high range, where $CV_{REF} = V_{DD}/4 + VR<3:0>/32 \times V_{DD}$.

With a 5 V supply, the available output range is from 0 V to 3.59 V.

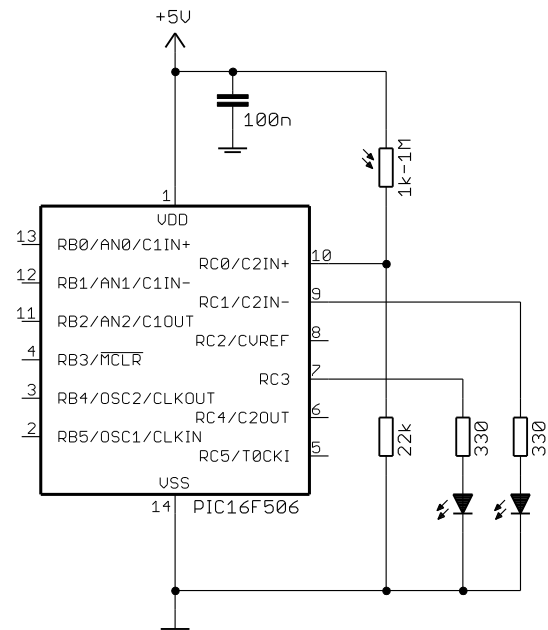
Since the low and high ranges overlap, only 29 of the 32 selectable voltages are unique ($0.250 \times V_{DD}$, $0.500 \times V_{DD}$ and $0.625 \times V_{DD}$ are selectable in both ranges).

The programmable voltage reference can optionally be output on the CVREF pin, whether or not it is also being used as the negative reference for comparator 2.

In [baseline assembler lesson 9](#), the circuit on the right was used to demonstrate how comparator 2 can be used with the programmable voltage reference to test whether an input signal is within an allowed band.

The C2IN+ input is used with a photocell to detect light levels, as before. The LED on RC3 indicates a low level of illumination, and the LED on RC1 indicates bright light. When neither LED is lit, the light level will be in the middle; not too dim or too bright.

If you are using the [Gooligum baseline training board](#), you should remove the shunt from of JP24 and instead place a shunt in position 2 ('C2IN+') of JP25, connecting photocell PH2 to C2IN+. You should also place shunts in JP17 and JP19, enabling the LEDs on RC1 and RC3.



To test whether the input is within limits, the programmable voltage reference is first configured to generate the “low” threshold voltage, and the input is compared with this low level. The voltage reference is then reconfigured to generate the “high” threshold and the input is compared with this higher level.

This process could be extended to multiple input thresholds, by configuring the voltage reference to generate each threshold in turn. However, if you wish to test against more than a few threshold levels, you would probably be better off using an analog-to-digital converter (described in the [next lesson](#)).

This example uses 2.0 V as the “low” threshold and 3.0 V as the “high” threshold, but, since the reference is programmable, you can always choose your own levels!

XC8

Comparator 2 can be configured in much the same way as we have been configuring comparator 1, either by assigning a binary value to CM2CON0:

```
CM2CON0 = 0b00101010;    // configure comparator 2:
//--1-----             normal polarity (C2POL = 1)
//-----1-             +ref is C2IN+ (C2PREF1 = 1)
//-----0--             -ref is CVref (C2NREF = 0)
//----1----             comparator on (C2ON = 1)
//                      -> C2OUT = 1 if C2IN+ > CVref
```

or by a block of statements assigning values to the bit-fields defined in the header files for CM2CON0:

```
// configure comparator 2
CM2CON0bits.C2PREF1 = 1;    // +ref is C2IN+
CM2CON0bits.C2NREF = 0;    // -ref is CVref
CM2CON0bits.C2POL = 1;    // normal polarity (C2IN+ > CVref)
CM2CON0bits.C2ON = 1;    // turn comparator on
// -> C2OUT = 1 if C2IN+ > CVref
```

Similarly, to configure the programmable voltage reference, we could assign a binary value to the variable corresponding to VRCON:

```
VRCON = 0b10000101;    // configure programmable voltage reference:
//1-----             enable voltage reference (VREN = 1)
//--0-0101             CVref = 0.406*Vdd (VRR = 0, VR = 5)
//                      -> CVref = 2.03 V
```

Or, using the bit-fields defined in the processor header files, we could write:

```
// configure voltage reference
VRCONbits.VRR = 0;           // CVref = 0.406*Vdd (2.03V) if VR = 5
                             //           or 0.594*Vdd (2.97V) if VR = 11
VRCONbits.VREN = 1;        // turn voltage reference on
```

We also need to assign a value between 0 and 15 to the VR<3:0> bits, to select the reference voltage.

Although they are available as single-bit fields VR0 to VR3, they are also (much more conveniently) made available as a 4-bit field named VR within VRCONbits, allowing us to simply write:

```
VRCONbits.VR = 5;           // CVref = 0.406*Vdd (2.03V)
```

Both of these examples selected a reference of $0.406 \times VDD$, giving $CVREF = 2.03 \text{ V}$ with a 5 V supply.

To generate a reference voltage close to 3.0 V, we can use:

```
VRCON = 0b10001011;       // configure programmable voltage reference:
//1-----               enable voltage reference (VREN = 1)
//--0-1011              CVref = 0.594*Vdd (VRR = 0, VR = 11)
//                       -> CVref = 2.97 V
```

or:

```
VRCONbits.VRR = 0;        // select high range
VRCONbits.VR = 11;       // CVref = 0.594*Vdd = 2.97 V
VRCONbits.VREN = 1;     // turn voltage reference on
```

Note that, by coincidence, the only difference between the settings for the two voltages is VR = 5 to select 2.03 V and VR = 11 to select 2.97 V.

This allows us to configure the other voltage reference settings just once, in the initialisation code:

```
// configure voltage reference
VRCONbits.VRR = 0;           // CVref = 0.406*Vdd (2.03V) if VR = 5
                             //           or 0.594*Vdd (2.97V) if VR = 11
VRCONbits.VREN = 1;        // turn voltage reference on
```

Then in the main loop we can switch between the two voltages, using:

```
VRCONbits.VR = 5;           // select low CVref (2.03 V)
```

and:

```
VRCONbits.VR = 11;         // select high CVref (2.97 V)
```

After changing the voltage reference, it can take a little while for it to settle and stably generate the newly-selected voltage. According to the data sheet, for the PIC16F506 this settling time can be up to 10 μs .

Therefore, we should insert a 10 μs delay after selecting a new voltage, before testing the comparator output.

As we saw in [lesson 1](#), we can do this using the ‘`__delay_us()`’ macro built into XC8.

Complete program

Here is how the above code fragments fit together, to form the complete “comparator 2 and programmable voltage reference demo” program for XC8:

```

/*****
*
*   Description:      Lesson 6, example 4
*
*   Demonstrates use of comparator 2 and programmable voltage reference
*
*   Turns on Low LED when C2IN+ < 2.0 V (low light level)
*                   or High LED when C2IN+ > 3.0 V (high light level)
*
*****/
*
*   Pin assignments:
*       C2IN+ = voltage to be measured (LDR/resistor divider)
*       RC3   = "Low" LED
*       RC1   = "High" LED
*
*****/

#include <xc.h>
#include <stdint.h>

#define _XTAL_FREQ 4000000 // oscillator frequency for delay functions

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & IOSCFCS_OFF & OSC_Intrc_RB4EN);

// Pin assignments
#define sLO      sPORTC.RC3 // "Low" LED (shadow)
#define sHI      sPORTC.RC1 // "High" LED (shadow)

/***** GLOBAL VARIABLES *****/
union { // shadow copy of PORTC
    uint8_t port;
    struct {
        unsigned RC0 : 1;
        unsigned RC1 : 1;
        unsigned RC2 : 1;
        unsigned RC3 : 1;
        unsigned RC4 : 1;
        unsigned RC5 : 1;
    };
} sPORTC;

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure ports
    TRISC = 0b110101; // configure RC1 and RC3 (only) as outputs

    // configure comparator 2
    CM2CON0bits.C2PREF1 = 1; // +ref is C2IN+

```



```

CM2CON0bits.C2NREF = 0;           // -ref is CVref
CM2CON0bits.C2POL = 1;           // normal polarity (C2IN+ > CVref)
CM2CON0bits.C2ON = 1;            // turn comparator on
                                   // -> C2OUT = 1 if C2IN+ > CVref

// configure voltage reference
VRCONbits.VRR = 0;               // CVref = 0.406*Vdd (2.03V) if VR = 5
                                   // or 0.594*Vdd (2.97V) if VR = 11
VRCONbits.VREN = 1;              // turn voltage reference on

/** Main loop
for (;;)
{
    // Test for low illumination
    VRCONbits.VR = 5;              // select low CVref (2.03 V)
    __delay_us(10);                // wait 10 us to settle
    sLO = !CM2CON0bits.C2OUT;      // if C2IN+ < CVref turn on Low LED

    // Test for high illumination
    VRCONbits.VR = 11;            // select high CVref (2.97 V)
    __delay_us(10);                // wait 10 us to settle
    sHI = CM2CON0bits.C2OUT;      // if C2IN+ > CVref turn on High LED

    // Display test results
    PORTC = sPORTC.port;          // copy shadow to PORTC
}
}

```

CCS PCB

As was alluded to earlier, the CCS PCB built-in `setup_comparator()` function is used to configure both comparators, not only comparator 1.

To configure comparator 2, OR one of the following symbols (defined in the “16F506.h” header file) into the expression passed to `setup_comparator()`:

```

#define CP2_B0_B1      0x30001C00
#define CP2_C0_B1      0x20100E00
#define CP2_C1_B1      0x20200C00
#define CP2_B0_VREF    0x10001800
#define CP2_C0_VREF    0x00100A00
#define CP2_C1_VREF    0x30200800

```

You’ll notice that there are more available input combinations than there were for comparator 1, and that for comparator 2, ‘VREF’ refers to the programmable voltage reference, instead of the 0.6 V reference.

You may also notice a mistake: **RB1** shares its pin with **C1IN-**, which is not available as an input to comparator 2. *CCS mistakenly included ‘B1’ in these symbols, when they should have written ‘C1’, referring to C2IN-, which shares its pin with RC1.*

If you do not include any of the comparator 1 configuration symbols in the expression, only comparator 2 will be setup, with comparator 1 being turned off. You can make this explicit by including the symbol ‘NC_NC’ into the expression.

For this example, we need:

```

setup_comparator(NC_NC|CP2_C0_VREF); // C1: off
                                   // C2: C2IN+ > CVref

```

To enable one of comparator 2's options, such as inverted polarity, output on C2OUT, or wake-up on change, OR one or more of these symbols into the `setup_comparator()` expression:

```
//Optionally OR with one or both of the following
#define CP2_OUT_ON_C4 0x00084000
#define CP2_INVERT    0x00002000
#define CP2_WAKEUP    0x00000100
```

To setup the programmable voltage reference, CCS provides another built-in function: `'setup_vref()'`.

It is used in this example as follows:

```
setup_vref(VREF_HIGH | 5); // CVref = 0.406*Vdd = 2.03 V
```

where '5' is the value the VR bits are being set to.

The first symbol is either `'VREF_LOW'` or `'VREF_HIGH'`, and specifies the voltage range you wish to select. It is ORed with a number between 0 and 15, which is loaded into `VR<3:0>`, to specify the voltage.

The symbol `'VREF_A2'` can optionally be ORed into the expression, to indicate that the reference voltage should be output on the CVREF pin.

Finally, as in the XC8 version, we should insert a 10 μ s delay after configuring the voltage reference, to allow it to settle.

This can be done with the built-in function `'delay_us()'`:

```
delay_us(10); // wait 10us to settle
```

Complete program

The following listing shows how these code fragments fit together in the CCS PCB version of the "comparator 2 and programmable voltage reference demo" program:

```
*****
* Description: Lesson 6, example 4 *
* * *
* Demonstrates use of Comparator 2 and programmable voltage reference *
* * *
* Turns on Low LED when C2IN+ < 2.0 V (low light level) *
* or High LED when C2IN+ > 3.0 V (high light level) *
* * *
*****
* Pin assignments: *
* C2IN+ = voltage to be measured (LDR/resistor divider) *
* RC3 = "Low" LED *
* RC1 = "High" LED *
* * *
*****/
```

```
#include <16F506.h>
```

```
#use delay (clock=4000000) // oscillator frequency for delay_ms()
```

```
/****** CONFIGURATION *****/
```

```
// ext reset, no code protect, no watchdog, 4 MHz int clock
```

```
#fuses MCLR,NOPROTECT,NOWDT,INTRC_IO,IOSC4
```

```

// Pin assignments
#define sLO      sPORTC.RC3      // "Low" LED (shadow)
#define sHI      sPORTC.RC1      // "High" LED (shadow)

/***** GLOBAL VARIABLES *****/
union {
    unsigned int8    port;      // shadow copy of PORTC
    struct {
        unsigned    RC0        : 1;
        unsigned    RC1        : 1;
        unsigned    RC2        : 1;
        unsigned    RC3        : 1;
        unsigned    RC4        : 1;
        unsigned    RC5        : 1;
    };
} sPORTC;

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure comparators
    setup_comparator(NC_NC|CP2_C0_VREF);    // C1: off
                                           // C2: C2IN+ > CVref

    /*** Main loop
    while (TRUE)
    {
        // Test for low illumination
        setup_vref(VREF_HIGH | 5);          // CVref = 0.406*Vdd = 2.03 V
        delay_us(10);                       // wait 10 us to settle
        sLO = ~C2OUT;                        // if C2IN+ < CVref turn on Low LED

        // Test for high illumination
        setup_vref(VREF_HIGH | 11);         // CVref = 0.594*Vdd = 2.97 V
        delay_us(10);                       // wait 10 us to settle
        sHI = C2OUT;                        // if C2IN+ > CVref turn on High LED

        // Display test results
        output_c(sPORTC.port);             // copy shadow to PORTC
    }
}

```

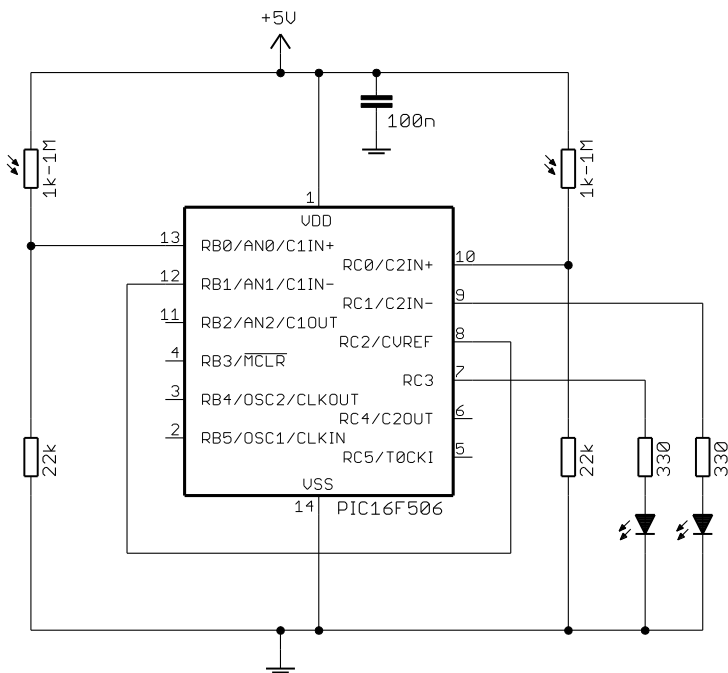
Using Both Comparators with the Programmable Voltage Reference

As a final example, suppose that we want to test two input signals (say, light level in two locations) by comparing them against a common reference. We would need to use two comparators, with an input signal connected to each, and a single threshold voltage level connected to both.

What if we want to use the programmable voltage reference to generate the common threshold?

We've seen that CVREF cannot be selected as an input to comparator 1, so it would seem that it's not possible to use the programmable voltage reference with comparator 1.

But although no internal connection is available, that doesn't rule out an external connection – and as we saw above, the programmable reference can be made available on the CVREF pin.



So, to use the programmable voltage reference with comparator 1, we need to set the VROE bit in the VRCON register, to enable the CVREF output, and connect the CVREF pin to a comparator 1 input – as shown in the circuit on the right, where CVREF is connected to C1IN-.

If you are using the [Gooligum baseline training board](#), you can keep the board set up as before, with shunts in JP17, JP19, and position 2 ('C2IN+') of JP25, and add a shunt across pins 2 and 3 ('LDR1') of JP24, to also connect photocell PH1 to C1IN+. You also need to connect CVREF to C1IN-, which you can do by linking pins 9 ('GP/RA/RB1') and 11 ('RC2') on the 16-pin header.

Note: You should disconnect your PICkit 2 or PICkit 3 from the board when you run the program (applying external power instead), because the programmer loads RB1/AN1/C1IN-, pulling down the reference voltage delivered by the CVREF pin.

XC8

Most of the initialisation and main loop code is very similar to that used in earlier examples, although setting up both comparators this time, but when configuring the voltage reference, we must ensure that the VROE bit is set, so that CVREF is available externally:

```
// configure voltage reference
VRCONbits.VRR = 1;           // select low range
VRCONbits.VR = 12;          // CVref = 0.500*Vdd
VRCONbits.VROE = 1;       // enable CVref output pin
VRCONbits.VREN = 1;         // turn voltage reference on
                             // -> CVref = 2.50 V (if Vdd = 5 V),
                             //   CVref output pin enabled
```

Complete program

Here is the complete XC8 version of the “two inputs with a common programmed voltage reference” program:

```
/*
 * Description: Lesson 6, example 5
 *
 * Demonstrates use of comparators 1 and 2
 * with the programmable voltage reference
 *
 * Turns on: LED 1 when C1IN+ > 2.5 V
 *           and LED 2 when C2IN+ > 2.5 V
 */
*****
```

```

*   Pin assignments:
*   C1IN+ = input 1 (LDR/resistor divider)
*   C1IN- = connected to CVref
*   C2IN+ = input 2 (LDR/resistor divider)
*   CVref = connected to C1IN-
*   RC1   = indicator LED 2
*   RC3   = indicator LED 1
*
*****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & IOSCFE_OFF & OSC_IntRC_RB4EN);

// Pin assignments
#define sLED1   sPORTC.RC3   // indicator LED 1
#define sLED2   sPORTC.RC1   // indicator LED 2

/***** GLOBAL VARIABLES *****/
union {
    uint8_t      port;           // shadow copy of PORTC
    struct {
        unsigned RC0      : 1;
        unsigned RC1      : 1;
        unsigned RC2      : 1;
        unsigned RC3      : 1;
        unsigned RC4      : 1;
        unsigned RC5      : 1;
    };
} sPORTC;

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure ports
    TRISC = 0b110101;           // configure RC1 and RC3 (only) as outputs

    // configure comparator 1
    CM1CON0bits.C1PREF = 1;     // +ref is C1IN+
    CM1CON0bits.C1NREF = 1;     // -ref is C1IN- (= CVref)
    CM1CON0bits.C1POL = 1;     // normal polarity (C1IN+ > C1IN-)
    CM1CON0bits.C1ON = 1;      // turn comparator on
    // -> C1OUT = 1 if C1IN+ > C1IN- (= CVref)

    // configure comparator 2
    CM2CON0bits.C2PREF1 = 1;    // +ref is C2IN+
    CM2CON0bits.C2NREF = 0;     // -ref is CVref
    CM2CON0bits.C2POL = 1;     // normal polarity (C2IN+ > CVref)
    CM2CON0bits.C2ON = 1;      // turn comparator on
    // -> C2OUT = 1 if C2IN+ > CVref

    // configure voltage reference
    VRCONbits.VRR = 1;         // select low range

```

```

VRCONbits.VR = 12;           // CVref = 0.500*Vdd
VRCONbits.VROE = 1;         // enable CVref output pin
VRCONbits.VREN = 1;         // turn voltage reference on
                             // -> CVref = 2.50 V (if Vdd = 5 V),
                             //   CVref output pin enabled

/** Main loop
for (;;)
{
    // start with shadow PORTC clear
    sPORTC.port = 0;

    // test comparator inputs
    sLED1 = CM1CON0bits.C1OUT; // turn on LED 1 if C1IN+ > CVref
    sLED2 = CM2CON0bits.C2OUT; // turn on LED 2 if C2IN+ > CVref

    // display test results
    PORTC = sPORTC.port;      // copy shadow to PORTC
}
}

```

CCS PCB

As we've seen, the 'setup_comparator()' function can be used to configure both comparators at once, so we can write:

```

// configure comparators
setup_comparator(CP1_B0_B1|CP2_C0_VREF); // C1: C1IN+ > C1IN-
                                           // C2: C2IN+ > CVref

```

To enable the CVREF pin when configuring the voltage reference, we need OR the symbol 'VREF_A2' into the expression passed to 'setup_vref()':

```

setup_vref(VREF_LOW | 12 | VREF_A2); // CVref = 0.500*Vdd = 2.50 V,
                                           // CVref output pin enabled

```

Complete program

Here is the complete CCS version of the "two inputs with a common programmed voltage reference" program:

```

/*****
*
* Description: Lesson 6, example 5
*
* Demonstrates use of comparators 1 and 2
* with the programmable voltage reference
*
* Turns on: LED 1 when C1IN+ > 2.5 V
*           and LED 2 when C2IN+ > 2.5 V
*
*****/
*
* Pin assignments:
* C1IN+ = input 1 (LDR/resistor divider)
* C1IN- = connected to CVref
* C2IN+ = input 2 (LDR/resistor divider)

```

```

*          CVref = connected to C1IN-          *
*          RC1  = indicator LED 2              *
*          RC3  = indicator LED 1              *
*                                                                 *
*****/

#include <16F506.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC_IO,IOSC4

// Pin assignments
#define sLED1  sPORTC.RC3      // indicator LED 1
#define sLED2  sPORTC.RC1      // indicator LED 2

/***** GLOBAL VARIABLES *****/
union {
    unsigned int8  port;          // shadow copy of PORTC
    struct {
        unsigned   RC0      : 1;
        unsigned   RC1      : 1;
        unsigned   RC2      : 1;
        unsigned   RC3      : 1;
        unsigned   RC4      : 1;
        unsigned   RC5      : 1;
    };
} sPORTC;

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure comparators
    setup_comparator(CP1_B0_B1|CP2_C0_VREF);    // C1 on: C1IN+ > C1IN-
                                                // C2 on: C2IN+ > CVref

    // configure voltage reference
    setup_vref(VREF_LOW | 12 | VREF_A2);        // CVref = 0.500*Vdd = 2.50 V,
                                                // CVref output pin enabled

    /*** Main loop
    while (TRUE)
    {
        // start with shadow PORTC clear
        sPORTC.port = 0;

        // test comparator inputs
        sLED1 = C1OUT;          // turn on LED 1 if C1IN+ > CVref
        sLED2 = C2OUT;          // turn on LED 2 if C2IN+ > CVref

        // display test results
        output_c(sPORTC.port); // copy shadow to PORTC
    }
}

```

Comparisons

Here is the resource usage summary for this example.

2xComp+VR

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	29	21	1
XC8 (Free mode)	34	52	1
CCS PCB	22	41	6

The CCS source code continues to be the shortest, being 50% shorter than of the XC8 source code – mainly because of the availability of built-in functions. But although the CCS compiler generates smaller code than the XC8 compiler (running in “Free mode”, with most optimisation disabled), the CCS-generated code is nevertheless nearly twice the size of the hand-written assembler version.

Summary

We have seen that it is possible to effectively utilise the comparators and voltage references available on baseline devices, such as the PIC16F506, using either the XC8 or CCS C compilers.

As we have come to expect, source code written for the CCS compiler is consistently concise, due to the availability of built-in functions.

However, we have also seen that, at least for the version of the compiler distributed with MPLAB⁴, CCS PCB lacks support for detecting wake-up on comparator change resets and that the symbols (defined in header files) for setting up the comparator 2 inputs are misleading.

On the whole, although the XC8 source code is longer, it’s more straightforward in some ways – if you’re familiar with the PIC registers. The CCS compiler lets you keep more distance from the detail of the internals, which as we’ve seen can be better or worse, depending on the situation.

But both approaches work!

The [next lesson](#) concludes our review of the baseline PIC architecture, covering analog to digital conversion and scaling and simple filtering of ADC readings for display (revisiting material from baseline lessons [10](#) and [11](#)).

⁴ As of August, 2012

Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 7: Analog-to-Digital Conversion and Simple Filtering

[Baseline assembler lesson 10](#) explained how to use the analog-to-digital converter (ADC) available on baseline PICs, such as the PIC16F506, using assembly language. This lesson demonstrates how to use C to control and access the ADC, re-implementing the examples using Microchip's XC8 (running in "Free mode") and CCS' PCB compilers¹.

It then shows how a simple moving-average filter, as described in [baseline assembler lesson 11](#), can be implemented in C. The final example implements a simple light meter, with the light level smoothed, scaled and shown as two decimal digits, using 7-segment LED displays.

In summary, this lesson covers:

- Configuring the ADC peripheral
- Reading analog inputs
- Hexadecimal output on 7-segment displays
- Working with arrays
- Accessing more than one bank of data memory
- Calculating a moving average to implement a simple filter

with examples for XC8 and CCS PCB.

Analog-to-Digital Converter

As explained in more detail in [baseline assembler lesson 10](#), the *analog-to-digital converter (ADC)* peripheral on baseline PICs allows analog input voltages to be measured, with a resolution of eight bits: 0 corresponds to VSS, and 255 corresponds to VDD.

The ADC module on the 16F506 has three external inputs, or *channels*: AN0, AN1 and AN2. Since there is only one ADC module, only one channel can be selected at one time, meaning that only one input can be read (*sampled* or *converted*) at once.

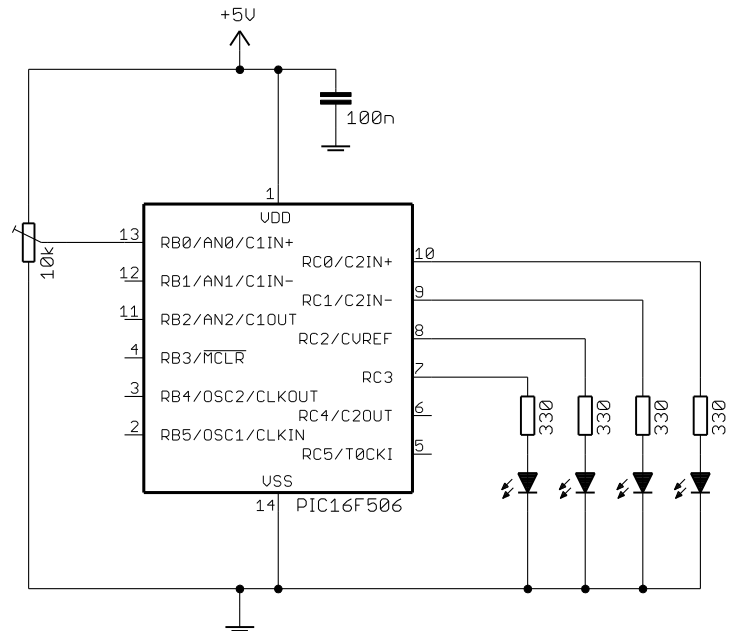
A simple example in [baseline lesson 10](#) demonstrated basic ADC operation, using use a potentiometer to provide a variable voltage to an analog input, and four LEDs to show a 4-bit binary representation of that value, using the circuit shown on the next page.

¹ XC8 is available as a free download from www.microchip.com, and CCS PCB is bundled for free with MPLAB 8

To implement it using the [Gooligum baseline training board](#), place a shunt across pins 1 and 2 ('POT') of JP24, connecting the 10 kΩ pot (RP2) to AN0, and shunts in JP16-19, enabling the LEDs on RC0-3.

If you are using Microchip's Low Pin Count Demo Board, the onboard pot and LEDs are already connected to AN0 and RC0 – RC3. You only need to ensure that jumpers JP1-5 are closed.

The voltage on AN0 is continually sampled, with the most significant four bits of the result being displayed on the LEDs, forming a 4-bit binary display.



The analog inputs share pins with RB0, RB1 and RB2. By default (after a power-on reset), the analog inputs are enabled. To use a pin for digital I/O, any analog function on that pin must first be disabled.

Whether a pin is configured for analog input is controlled by the ANS<1:0> bits in the ADCON0 register, as shown on in the table on the right.

The pins cannot be configured independently; only the listed combinations are possible.

A quick way to disable the analog inputs is to clear ADCON0, since clearing ANS<1:0> deselects all the analog inputs.

ANS<1:0>	Pins configured as analog inputs
00	none
01	AN2 only
10	AN0 and AN2
11	AN0, AN1 and AN2

In this example, only AN0 has to be configured as an analog input; either of the combinations which include AN0 could be used – in this case, the “AN0 and AN2” option, selected by ANS<1:0> = ‘10’, is used.

The appropriate ADC input channel must also be selected. This is controlled by the CHS<1:0> bits in ADCON0, as shown on the right.

Note that, in addition to the three external analog inputs, the 0.6 V fixed voltage reference is selectable as an ADC channel. We'll use this feature in a later example.

In this example, AN0 has to be selected as the ADC channel, specified by CHS<1:0> = ‘00’.

CHS<1:0>	ADC channel
00	analog input AN0
01	analog input AN1
10	analog input AN2
11	0.6 V internal voltage reference

An appropriate ADC conversion clock source must be selected, specified by the ADCS<1:0> bits in ADCON0. As explained in [baseline assembler lesson 10](#), the INTOSC/4 clock option (ADCS<1:0> = ‘11’) is a safe option which will always work, so that option is used here.

Finally, the ADC peripheral must be turned on, by setting the ADON bit (in ADCON0) to ‘1’.

In the example in [baseline assembler lesson 10](#), the ADC was configured with the above options with:

```
movlw    b'10110001'    ; configure ADC:
          ; 10-----    AN0, AN2 analog (ANS = 10)
          ; --11-----  clock = INTOSC/4 (ADCS = 11)
          ; ----00--    select channel AN0 (CHS = 00)
          ; -----1    turn ADC on (ADON = 1)
movwf    ADCON0         ; -> AN0 ready for sampling
```

To begin a conversion, the GO/\overline{DONE} bit (in `ADCON0`) is set:

```
bsf      ADCON0,GO      ; start conversion
```

It is then necessary to wait until the GO/\overline{DONE} bit is clear:

```
w_adc   btfsc   ADCON0,NOT_DONE ; wait until done
        goto    w_adc
```

The result of the conversion is then available in the `ADRES` register:

```
swapf   ADRES,w        ; copy high nybble of result
movwf   PORTC          ; to low nybble of output port (LEDs)
```

Note that, in this example, the most significant four bits of the result are copied to the least four significant bits of `PORTC`, because the LEDs are connected to `RC0 – RC3`.

We saw in [baseline assembler lesson 10](#) that, to use `RC0` and `RC1` for digital I/O, the `C2IN+` and `C2IN-` inputs must be disabled. This was done by clearing `CM2CON0`:

```
clrf    CM2CON0        ; disable comparator 2 -> RC0, RC1 digital
```

We also saw that, to use `RC2` for digital I/O, the `CVREF` output has to be disabled. Although the programmable voltage reference module is disabled by default, it was explicitly turned off in the example, by clearing `VRCON`:

```
clrf    VRCON          ; disable CVref -> RC2 usable
```

XC8

Since `XC8` makes the special function registers directly accessible through variables defined in the device-specific header files, the code to configure `RC0 – RC3` as outputs is simply:

```
// configure ports
TRISC = 0b110000;          // configure RC0-RC3 as outputs
CM2CON0 = 0;              // disable comparator 2 -> RC0, RC1 digital
VRCON = 0;                // disable CVref -> RC2 usable
```

Configuring the ADC module could then be done in the same way, by assigning a value to `ADCON0`:

```
// configure ADC
ADCON0 = 0b10110001;
          //10-----    AN0, AN2 analog (ANS = 10)
          //--11-----  clock = INTOSC/4 (ADCS = 11)
          //----00--    select channel AN0 (CHS = 00)
          //-----1    turn ADC on (ADON = 1)
```

However, as we have seen in the earlier lessons, the XC8 header files define most special function registers, including `ADCON0`, as unions of structures containing bit-fields corresponding to that register's bits.

Thus, we can configure the ADC module with:

```
// configure ADC
ADCON0bits.ADCS = 0b11;    // clock = INTOSC/4
ADCON0bits.ANS  = 0b10;    // AN0, AN2 analog
ADCON0bits.CHS  = 0b00;    // select channel AN0
ADCON0bits.ADON = 1;       // turn ADC on
                          // -> AN0 ready for sampling
```

Although this approach involves more statements, leading to a longer program and a larger executable, it has the advantage of clarity, is less prone to errors, and seems more “natural” when programming in C – so it's the method we'll use in the examples in this lesson. But as ever, which approach you use is a question of personal programming style – they're both valid.

Like MPASM, the XC8 device headers define more than one symbol for the `GO/DONE` bit.

In fact you can access it as any of:

```
ADCON0bits.GO_nDONE
ADCON0bits.GO
ADCON0bits.nDONE
```

As we did in [baseline assembler lesson 10](#), we'll use the “GO” bit-field when starting the conversion:

```
ADCON0bits.GO = 1;        // start conversion
```

and we'll use the “nDONE” version of the bit-field when waiting for the conversion to finish:

```
while (ADCON0bits.nDONE) // wait until done
    ;
```

even though they are referring to the same bit – the intent of the code is clearer this way.

The result of the conversion is available in `ADRES`, accessible through the ‘`ADRES`’ variable.

We need to copy the upper four bits of the result to the lower four bits of `PORTC` (where the LEDs are connected). This means shifting the result four bits to the right, so we can write simply:

```
LEDS = ADRES >> 4;       // copy high nybble of result to LEDs
```

(having defined ‘`LEDS`’ as an alias for ‘`PORTC`’)

Complete program

Here is how the above code fragments fit together:

```
/******
 *
 * Description:    Lesson 7, example 1
 *
 * Demonstrates basic use of ADC
 *
 * Continuously samples analog input, copying value to 4 x LEDs
 *
 *****/
```

```

*****
*
*   Pin assignments:
*       AN0       = voltage to be measured (e.g. pot output)
*       RC0-3     = output LEDs (RC3 is MSB)
*
*****/

#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & IOSCFSS_OFF & OSC_IntrRC_RB4EN);

// Pin assignments
#define LEDES      PORTC          // output LEDs on RC0-RC3

/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure ports
    TRISC = 0b110000;             // configure RC0-RC3 as outputs
    CM2CON0 = 0;                 // disable comparator 2 -> RC0, RC1 digital
    VRCON = 0;                   // disable CVref -> RC2 usable

    // configure ADC
    ADCON0bits.ADCS = 0b11;     // clock = INTOSC/4
    ADCON0bits.ANS  = 0b10;     // AN0, AN2 analog
    ADCON0bits.CHS  = 0b00;     // select channel AN0
    ADCON0bits.ADON = 1;        // turn ADC on
                                // -> AN0 ready for sampling

    //*** Main loop
    for (;;)
    {
        // sample analog input
        ADCON0bits.GO = 1;       // start conversion
        while (ADCON0bits.nDONE) // wait until done
            ;

        // display result on 4 x LEDs
        LEDES = ADRES >> 4;      // copy high nybble of result to LEDs
    }
}

```

CCS PCB

We saw in the [lesson 6](#) that the CCS compiler provides a built-in function, 'setup_comparator()', which can be used to disable comparator 2 (so that we can use RC0 and RC1 as digital outputs):

```
setup_comparator(NC_NC_NC_NC); // disable comparators -> RC0, RC1 digital
```

Note that this command actually disables both comparators, but since comparator 1 is not used in this example, there is no reason to enable it.

Similarly, the `setup_vref()` function can be used to disable the CVREF output, making RC2 usable:

```
setup_vref(FALSE); // disable CVref -> RC2 usable
```

A number of built-in functions are used to configure the ADC module.

The `setup_adc_ports()` function is used to select which ports are configured as analog inputs.

It is called with one of the symbols defined in the device's header file. For example, "16F506.h" contains:

```
// Constants used in SETUP_ADC_PORTS() are:
#define AN0_AN1_AN2      0xc0 // A0 A1 A2
#define AN0_AN2          0x80 // A0 A2
#define AN2              0x40 // A2
#define NO_ANALOGS      0 // None
```

In this case, we want the AN0 and AN2 configuration, so we use:

```
setup_adc_ports(AN0_AN2); // configure AN0 and AN2 for analog input
```

Note that, if you wanted to disable all the analog inputs, you would use:

```
setup_adc_ports(NO_ANALOGS); // no analog inputs (all digital)
```

The `setup_adc()` function is used to select the ADC clock source, or to turn the ADC module off (useful for saving power in sleep mode).

It is also called with a symbol defined in the device's header file. For example, "16F506.h" contains:

```
// Constants used for SETUP_ADC() are:
#define ADC_OFF          0 // ADC Off
#define ADC_CLOCK_DIV_32 0x00
#define ADC_CLOCK_DIV_16 0x10
#define ADC_CLOCK_DIV_8  0x20
#define ADC_CLOCK_INTERNAL 0x30 // Internal 2-6us
```

In this case we want the internal clock source, so we use:

```
setup_adc(ADC_CLOCK_INTERNAL); // select INTOSC/4 clock and turn ADC on
```

Note that the ADC is implicitly being turned on by this function. If you don't want it turned on, you need to explicitly turn it off, with:

```
setup_adc(ADC_OFF); // turn ADC module off
```

The `set_adc_channel()` function is used to select the ADC input channel.

The parameter corresponds to the value of the CHS channel selection bits, as defined in the device data sheet (and, for the 16F506, in the table above).

In this case, we want channel 0, corresponding to AN0, so we use:

```
set_adc_channel(0); // ADC channel = AN0
```

Initiating the conversion, waiting for it to complete, then returning the result can be done with a single built-in function: `read_adc()`.

It can optionally be passed one of the symbols defined in the header file, for example:

```
// Constants used in READ_ADC() are:
#define ADC_START_AND_READ    7 // This is the default if nothing is specified
#define ADC_START_ONLY       1
#define ADC_READ_ONLY        6
```

This means that you can start a conversion with:

```
read_adc(ADC_START_ONLY); // start ADC conversion
```

and do something else while waiting for the conversion to complete (indicated by the 'adc_done()' built-in function), and then read the result with something like:

```
result = read_adc(ADC_READ_ONLY); // read ADC result
```

In this case, we want to initiate the conversion and then read the result in a single operation, so to sample the input and place the upper four bits of the result in the lower four bits of PORTC, we can write:

```
output_c(read_adc()>>4); // read ADC and copy high nybble of result to LEDs
```

Note that there is no need to specify 'ADC_START_AND_READ' as the parameter to 'read_adc()', since it is the default if nothing is specified.

Complete program

Here is how these code fragments fit together in the CCS version of the "4 LEDs ADC demo" program:

```
/******
 *
 * Description: Lesson 7, example 1
 *
 * Demonstrates basic use of ADC
 *
 * Continuously samples analog input, copying value to 4 x LEDs
 *
 *****/
 *
 * Pin assignments:
 * AN0 = voltage to be measured (e.g. pot output or LDR)
 * RC0-3 = output LEDs (RC3 is MSB)
 *
 *****/
```

```
#include <16F506.h>
```

```
/****** CONFIGURATION *****/
```

```
// ext reset, no code protect, no watchdog, 4 MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC_IO,IOSC4
```

```
/****** MAIN PROGRAM *****/
```

```
void main()
```

```
{
```

```
    //*** Initialisation
```

```
    // configure ports
```

```
    setup_comparator(NC_NC_NC_NC); // disable comparators -> RC0, RC1 digital
```

```
    setup_vref(FALSE); // disable CVref -> RC2 usable
```

```

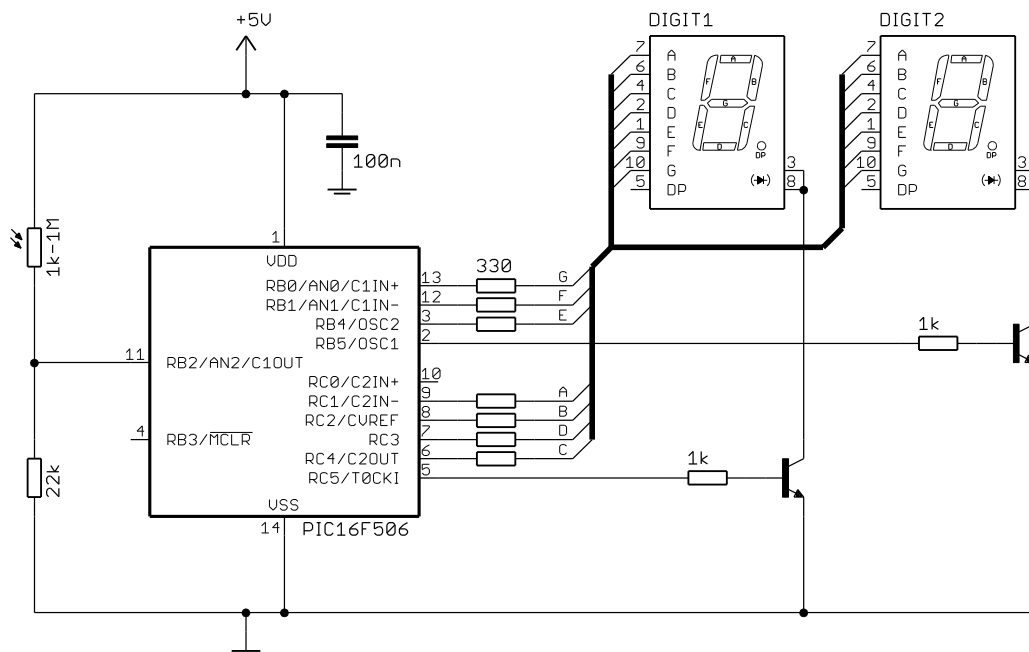
// configure ADC
setup_adc(ADC_CLOCK_INTERNAL); // clock = INTOSC/4, turn ADC on
setup_adc_ports(AN0_AN2); // AN0, AN2 analog
set_adc_channel(0); // select channel AN0
// -> AN0 ready for sampling

/** Main loop
while (TRUE)
{
    // sample and display analog input
    output_c(read_adc() >> 4); // read ADC and copy result to LEDs
}
}

```

Hexadecimal Output

To add a more useful, human-readable output to the ADC demo, the second example in [baseline assembler lesson 10](#) implemented a two-digit hexadecimal display, based on the multiplexed 7-segment display circuit from [baseline assembler lesson 8](#), dropping one digit, and adding a photocell and resistor to supply a voltage that increases with light level, as shown below:



To implement this circuit using the [Gooligum baseline training board](#), place shunts:

- across every position (all six of them) of jumper block JP4, connecting segments A-D, F and G to pins RB0-1 and RC1-4
- in position 1 ('RA/RB4') of JP5, connecting segment E to pin RB4
- across pins 2 and 3 ('RC5') of JP6, connecting digit 1 to the transistor controlled by RC5
- in jumpers JP8 and JP9, connecting pins RC5 and RB5 to their respective transistors
- in position 1 ('AN2') of JP25, connecting photocell PH2 to AN2.

All other shunts should be removed.

The source code was also adapted from the timer-based 7-segment display multiplexing routines presented in [baseline assembler lesson 8](#), with the only important differences being:

- the value to be displayed was now the result of an analog-to-digital conversion, performed using the code from the first example (above), instead of a time count;
- the pattern lookup table for the 7-segment display was extended from 10 to 16 entries, to include representations of the letters 'A' to 'F';

XC8

The previous example included initialisation code to disable comparator 2 and the programmable voltage reference. Extending this to also disable comparator 1 is simply:

```
CM1CON0 = 0;           // disable comparator 1 -> RB0, RB1 digital
CM2CON0 = 0;           // disable comparator 2 -> RC0, RC1 digital
VRCON = 0;             // disable CVref -> RC2 usable
```

We also need to configure the ADC, but this time with AN2 as the only analog input:

```
// configure ADC
ADCON0bits.ADCS = 0b11;           // clock = INTOSC/4
ADCON0bits.ANS = 0b01;           // AN2 (only) analog
ADCON0bits.CHS = 0b10;           // select channel AN2
ADCON0bits.ADON = 1;             // turn ADC on
// -> AN2 ready for sampling
```

The ADC input is sampled, using code from the previous example:

```
// sample input
ADCON0bits.GO = 1;               // start conversion
while (ADCON0bits.nDONE)        // wait until done
    ;
```

Then the result is displayed, using code adapted from [lesson 5](#):

```
// display high nybble for 2.048 ms
while (!TMR0_2)                  // wait for TMR0<2> to go high
    ;
set7seg(ADRES >> 4);             // output high nybble of result
TENS_EN = 1;                     // enable "tens" digit
while (TMR0_2)                  // wait for TMR0<2> to go low
    ;

// display low nybble for 2.048 ms
while (!TMR0_2)                  // wait for TMR0<2> to go high
    ;
set7seg(ADRES & 0x0F);           // output low nybble of result
ONES_EN = 1;                     // enable ones digit
while (TMR0_2)                  // wait for TMR0<2> to go low
    ;
```

The 'set7seg()' function is much the same as that presented in [lesson 5](#), but with the pattern arrays (lookup tables) now extended from 10 to 16 entries, adding the 7-segment representations of the letters 'A' to 'F'.

Complete program

Here is the complete XC8 version of the “ADC demo with hexadecimal output” program, showing how these code fragments – mostly adapted from previous programs – fit together:

```

/*****
*   Description:      Lesson 7, example 2
*
*   Displays ADC output in hexadecimal on 7-segment LED displays
*
*   Continuously samples analog input,
*   displaying result as 2 x hex digits on multiplexed 7-seg displays
*
*****/
*
*   Pin assignments:
*       AN2          = voltage to be measured (e.g. pot or LDR)
*       RB0-1,RB4,RC1-4 = 7-segment display bus (common cathode)
*       RC5          = "tens" digit enable (active high)
*       RB5          = ones digit enable
*
*****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & IOSCFSS_OFF & OSC_IntrC_RB4EN);

// Pin assignments
#define TENS_EN      PORTCbits.RC5    // "tens" (high nybble) digit enable
#define ONES_EN     PORTBbits.RB5    // ones digit enable

/***** PROTOTYPES *****/
void set7seg(uint8_t digit);    // display digit on 7-segment display

/***** MACROS *****/
#define TMR0_2      (TMR0 & 1<<2)    // access to TMR0<2>

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure ports
    TRISB = 0;                // configure PORTB and PORTC as all outputs
    TRISC = 0;
    CM1CON0 = 0;              // disable comparator 1 -> RB0, RB1 digital
    CM2CON0 = 0;              // disable comparator 2 -> RC0, RC1 digital
    VRCON = 0;                // disable CVref -> RC2 usable

    // configure ADC
    ADCON0bits.ADCS = 0b11;   // clock = INTOSC/4
    ADCON0bits.ANS = 0b01;    // AN2 (only) analog
    ADCON0bits.CHS = 0b10;    // select channel AN2
    ADCON0bits.ADON = 1;      // turn ADC on
                                // -> AN2 ready for sampling

```

```

// configure timer
OPTION = 0b11010111;
    //--0-----
    //----0---
    //-----111
    //
    //
// configure Timer0:
    timer mode (T0CS = 0) -> RC5 usable
    prescaler assigned to Timer0 (PSA = 0)
    prescale = 256 (PS = 111)
    -> increment every 256 us
    (TMR0<2> cycles every 2.048 ms)

/***/ Main loop
for (;;)
{
    // sample input
    ADCON0bits.GO = 1;           // start conversion
    while (ADCON0bits.nDONE)    // wait until done
        ;

    // display high nybble for 2.048 ms
    while (!TMR0_2)             // wait for TMR0<2> to go high
        ;
    set7seg(ADRES >> 4);        // output high nybble of result
    TENS_EN = 1;                // enable "tens" digit
    while (TMR0_2)             // wait for TMR0<2> to go low
        ;

    // display low nybble for 2.048 ms
    while (!TMR0_2)            // wait for TMR0<2> to go high
        ;
    set7seg(ADRES & 0x0F);      // output low nybble of result
    ONES_EN = 1;                // enable ones digit
    while (TMR0_2)             // wait for TMR0<2> to go low
        ;
}
}

/***** FUNCTIONS *****/

/***** Display digit on 7-segment display *****/
void set7seg(uint8_t digit)
{
    // pattern table for 7 segment display on port B
    const uint8_t pat7segB[16] = {
        // RB4 = E, RB1:0 = FG
        0b010010, // 0
        0b000000, // 1
        0b010001, // 2
        0b000001, // 3
        0b000011, // 4
        0b000011, // 5
        0b010011, // 6
        0b000000, // 7
        0b010011, // 8
        0b000011, // 9
        0b010011, // A
        0b010011, // b
        0b010010, // C
        0b010001, // d
        0b010011, // E
        0b010011 // F
    };
};

```

```

// pattern table for 7 segment display on port C
const uint8_t pat7segC[16] = {
    // RC4:1 = CDBA
    0b011110, // 0
    0b010100, // 1
    0b001110, // 2
    0b011110, // 3
    0b010100, // 4
    0b011010, // 5
    0b011010, // 6
    0b010110, // 7
    0b011110, // 8
    0b011110, // 9
    0b010110, // A
    0b011000, // b
    0b001010, // C
    0b011100, // d
    0b001010, // E
    0b000010 // F
};

// disable displays
PORTB = 0; // clear all digit enable lines on PORTB
PORTC = 0; // and PORTC

// output digit pattern
PORTB = pat7segB[digit]; // lookup and output port B and C patterns
PORTC = pat7segC[digit];
}

```

CCS PCB

Since the built-in `setup_comparator()` function can be used to disable both comparators with a single call, the code to disable the comparators and the voltage reference is the same as in the first example, above:

```

setup_comparator(NC_NC_NC_NC); // disable comps -> RB0-1, RC0-1 digital
setup_vref(FALSE); // disable CVref -> RC2 usable

```

In this example, the ADC has to be configured with AN2 as the only analog input:

```

setup_adc(ADC_CLOCK_INTERNAL); // clock = INTOSC/4, turn ADC on
setup_adc_ports(AN2); // AN2 (only) analog
set_adc_channel(2); // select channel AN2

```

Because we need to access the ADC result twice (once for each digit in the display), it makes sense to sample the input and store the result in a variable, for later reference:

```

adc_res = read_adc();

```

This result is then displayed, using code adapted from [lesson 5](#):

```

// display high nybble for 2.048 ms
while (!TMR0_2) // wait for TMR0<2> to go high
;
set7seg(adc_res >> 4); // output high nybble of result
output_high(TENS_EN); // enable "tens" digit

```

```

while (TMR0_2)                // wait for TMR0<2> to go low
    ;

// display low nybble for 2.048 ms
while (!TMR0_2)               // wait for TMR0<2> to go high
    ;
set7seg(adc_res & 0x0F);      // output low nybble of result
output_high(ONES_EN);        // enable ones digit
while (TMR0_2)                // wait for TMR0<2> to go low
    ;

```

Note that, instead of storing the ADC result in a variable, we could have written:

```

// display high nybble for 2.048 ms
while (!TMR0_2)               // wait for TMR0<2> to go high
    ;
set7seg(read_adc() >> 4);     // sample input, then
                                // output high nybble of result
output_high(TENS_EN);         // enable "tens" digit
while (TMR0_2)                // wait for TMR0<2> to go low
    ;

// display low nybble for 2.048 ms
while (!TMR0_2)               // wait for TMR0<2> to go high
    ;
set7seg(read_adc(ADC_READ_ONLY) & 0x0F); // output low nybble of result
output_high(ONES);            // enable ones digit
while (TMR0_2)                // wait for TMR0<2> to go low
    ;

```

This uses the ‘`read_adc()`’ function to sample the input as part of the first digit display routine, and then uses the ‘`read_adc(ADC_READ_ONLY)`’ form of the function to return the already-sampled result, when displaying the second digit. However, although this approach saves a line of code and avoids the need to allocate a variable, it seems a little unwieldy. Again, it’s really a question of personal style.

As in the XC8 example, the ‘`set7seg()`’ function is much the same as that presented in [lesson 5](#), but with the pattern arrays extended from 10 to 16 entries.

Complete program

Here is the complete CCS version of the “ADC demo with hexadecimal output” program, showing how these code fragments – again mostly adapted from previous programs – fit together:

```

/*****
*
* Description:    Lesson 7, example 2
*
* Displays ADC output in hexadecimal on 7-segment LED displays
*
* Continuously samples analog input,
* displaying result as 2 x hex digits on multiplexed 7-seg displays
*
*****
*
* Pin assignments:
*     AN2                = voltage to be measured (e.g. pot or LDR)
*     RB0-1,RB4,RC1-4    = 7-segment display bus (common cathode)
*

```

```

*          RC5          = "tens" digit enable (active high)          *
*          RB5          = ones digit enable                          *
*                                                                *
*****/

#include <16F506.h>

/***** CONFIGURATION *****/
// Config: ext reset, no code protect, no watchdog, 4 MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC_IO,IOSC4

// Pin assignments
#define TENS_EN    PIN_C5          // "tens" (high nybble) enable
#define ONES_EN    PIN_B5          // ones enable

/***** PROTOTYPES *****/
void set7seg(unsigned int8 digit);    // display digit on 7-segment display

/***** MACROS *****/
#define TMR0_2    (get_timer0() & 1<<2)    // access to TMR0<2>

/***** MAIN PROGRAM *****/
void main()
{
    unsigned int8    adc_res;          // result of ADC conversion

    //*** Initialisation

    // configure ports
    setup_comparator(NC_NC_NC_NC);    // disable compss -> RB0-1, RC0-1 digital
    setup_vref(FALSE);                // disable CVref -> RC2 usable

    // configure ADC
    setup_adc(ADC_CLOCK_INTERNAL);    // clock = INTOSC/4, turn ADC on
    setup_adc_ports(AN2);              // AN2 (only) analog
    set_adc_channel(2);                // select channel AN2
                                        // -> AN2 ready for sampling

    // configure Timer0
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_256); // timer mode, prescale = 256
                                                // -> bit 2 cycles every 2.048 ms

    //*** Main loop
    while (TRUE)
    {
        // sample input
        adc_res = read_adc();

        // display high nybble for 2.048 ms
        while (!TMR0_2)                // wait for TMR0<2> to go high
            ;
        set7seg(adc_res >> 4);          // output high nybble of result
        output_high(TENS_EN);           // enable "tens" digit
        while (TMR0_2)                 // wait for TMR0<2> to go low
            ;

        // display low nybble for 2.048 ms
    }
}

```

```

        while (!TMR0_2)                // wait for TMR0<2> to go high
        ;
        set7seg(adc_res & 0x0F);       // output low nybble of result
        output_high(ONES_EN);         // enable ones digit
        while (TMR0_2)                // wait for TMR0<2> to go low
        ;
    }
}

/***** FUNCTIONS *****/

/***** Display digit on 7-segment display *****/
void set7seg(unsigned int8 digit)
{
    // pattern table for 7 segment display on port B
    const int8 pat7segB[16] = {
        // RB4 = E, RB1:0 = FG
        0b010010, // 0
        0b000000, // 1
        0b010001, // 2
        0b000001, // 3
        0b000011, // 4
        0b000011, // 5
        0b010011, // 6
        0b000000, // 7
        0b010011, // 8
        0b000011, // 9
        0b010011, // A
        0b010011, // b
        0b010010, // C
        0b010001, // d
        0b010011, // E
        0b010011 // F
    };

    // pattern table for 7 segment display on port C
    const int8 pat7segC[16] = {
        // RC4:1 = CDBA
        0b011110, // 0
        0b010100, // 1
        0b001110, // 2
        0b011110, // 3
        0b010100, // 4
        0b011010, // 5
        0b011010, // 6
        0b010110, // 7
        0b011110, // 8
        0b011110, // 9
        0b010110, // A
        0b011000, // b
        0b001010, // C
        0b011100, // d
        0b001010, // E
        0b000010 // F
    };

    // disable displays
    output_b(0); // clear all digit enable lines on PORTB
    output_c(0); // and PORTC
}

```

```

// output digit pattern
output_b(pat7segB[digit]); // lookup and output port B and C patterns
output_c(pat7segC[digit]);
}
    
```

Comparisons

Here is the resource usage for the “ADC demo with hexadecimal output” assembler and C examples:

ADC_hex-out

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	96	86	1
XC8 (Free mode)	68	161	2
CCS PCB	63	135	8

Despite the different approaches of the two C compilers (direct register access versus built-in functions), the source code written for XC8 is much the same length as that for CCS PCB, and around two thirds the length of the assembler source. On the other hand, the optimised code generated by the CCS compiler is more than 50% larger than the assembler version.

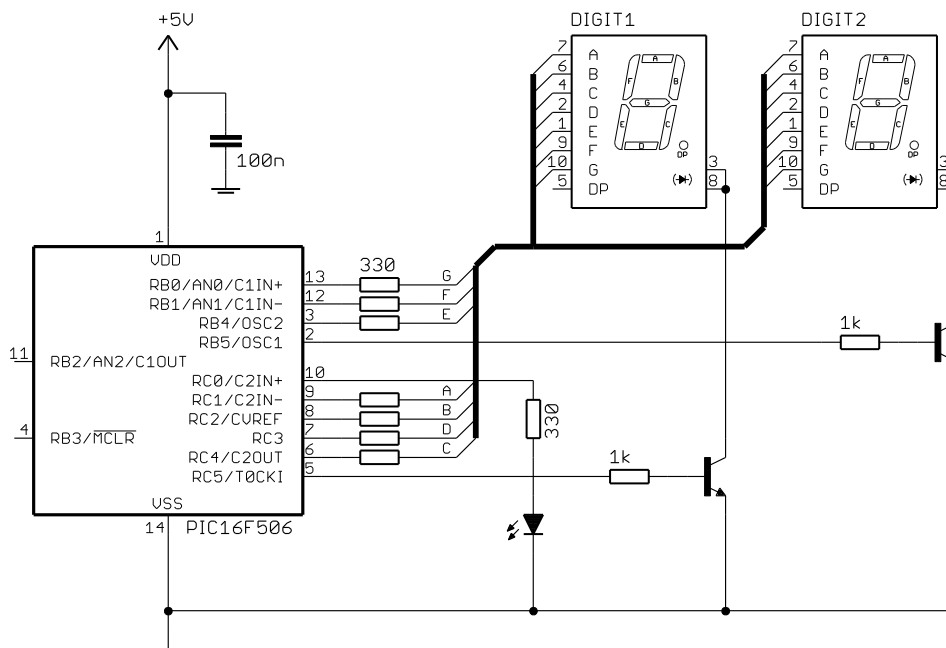
Measuring Supply Voltage

The fact that the absolute 0.6 V reference can be selected as an ADC input channel means that it can be used to infer the supply voltage (effectively VDD, given that in most cases VSS = 0 V), since the 0.6 V reference will read as $0.6\text{ V} \div VDD \times 255$.

For VDD = 5.0 V, the expected ADC result is $0.6\text{ V} \div 5.0\text{ V} \times 255 = 30$.

As VDD falls, the ADC reading corresponding to 0.6 V rises. This gives us a way to check that the power supply voltage (perhaps from a battery) is adequate, and to shut down the circuit and/or provide a warning if it falls too low.

The circuit shown below was used in [baseline assembler lesson 10](#) to demonstrate this.



If you are using the [Gooligum baseline training board](#), you should set it up as in the last example, but remove the shunt from JP25 (disconnecting the photocell from AN2) and close JP16 (connecting the LED on RC0).

As in the last example, the ADC result (now representing the value of the 0.6 V reference) is displayed in hex on the 7-segment displays, but to indicate low voltage, the LED on RC0 is lit if VDD falls below 3.5 V.

XC8

Most of the program code is the same as that in the previous example, but because we are now sampling the internal 0.6 V reference instead of AN0, the ADC has to be configured differently:

```
// configure ADC
ADCON0bits.ADCS = 0b11;           // clock = INTOSC/4
ADCON0bits.ANS  = 0b00;           // no analog inputs -> RB0-2 digital
ADCON0bits.CHS  = 0b11;           // select 0.6 V reference
ADCON0bits.ADON = 1;              // turn ADC on
                                   // -> 0.6 V reference ready for sampling
```

The code to sample the ADC and output the result on the 7-segment displays is the same as before, but we need to add some code to test for the under-voltage condition ($V_{DD} < 3.5$ V).

In the assembler example, the minimum allowable VDD was defined as a constant at the beginning of the program, so that it could be easily changed later:

```
constant MINVDD=3500                ; Minimum Vdd (in mV)
```

It was necessary to express this as an integer, because MPASM does not support floating-point expressions. Thus, the expression to convert this minimum VDD value to a constant which could be used to compare the ADC result with also had to be written using only integers:

```
constant VRMAX=255*600/MINVDD      ; Threshold for 0.6V ref measurement
```

Since C does support floating-point expressions, it is tempting to define the minimum VDD as a floating-point constant:

```
#define MINVDD 3.5                  // minimum Vdd (Volts)
```

and to then write the ADC comparison as:

```
if (ADRES > 0.6/MINVDD*255)        // if measured 0.6V > threshold
    WARN = 1;                       // light warning LED
```

Writing it that way makes the code very clear, because we normally refer to the internal reference as 0.6 V, not 600 mV, and it is natural to express the minimum VDD as 3.5 V, not 3500 mV.

But there is a big problem with this – and it is a very easy mistake to make, when using C with small microcontrollers. The compiler sees ‘0.6/MINVDD*255’ as being a floating-point expression (which, of course, it is), and implements the comparison as a floating-point operation. To do so, it links a number of floating-point routines into the code, and generates code to convert ADRES into floating-point form, passing it to a floating-point comparison routine. This greatly increases the size of the generated code, blowing out to 508 words of program memory²! Compare this with the previous example, which is almost identical –

² using XC8 v1.01 running in ‘Free mode’

lacking only this comparison routine – but required only 161 words of program memory. You wouldn't expect that adding such a simple routine would more than triple the size of the generated program! And normally it wouldn't; the only reason the generated code is so large is that floating-point routines have been inadvertently, and unnecessarily, included into it.

Note: The inadvertent use of floating-point expressions in C programs can lead the C compiler to unnecessarily link floating-point routines into the object code, significantly increasing the size of the generated code.

There are a number of ways to overcome this problem, including the use of integer-only expressions, but surely the simplest method, while maintaining clarity, is to explicitly *cast* the expression as an integer:

```
if (ADRES > (int)(0.6/MINVDD*255)) // if measured 0.6 V > threshold
    WARN = 1;                       // light warning LED
```

This simple change prevents the compiler from including floating-point code, reducing the size of the generated code from 508 to only 165 words of program memory!

Program listing

The only change to the program setup (device configuration, function prototypes etc.) from the previous example is the addition of the following constant definition:

```
/****** CONSTANTS *****/
#define MINVDD 3.5 // minimum Vdd (Volts)
```

Most of the rest of the source code is identical to the previous example, but it is worth looking at the main program code, so that you can see the new ADC configuration and how the comparison code fits into the sample and display loop:

```
/****** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure ports
    TRISB = 0; // configure PORTB and PORTC as all outputs
    TRISC = 0;
    CM1CON0 = 0; // disable comparator 1 -> RB0, RB1 digital
    CM2CON0 = 0; // disable comparator 2 -> RC0, RC1 digital
    VRCON = 0; // disable CVref -> RC2 usable

    // configure ADC
    ADCON0bits.ADCS = 0b11; // clock = INTOSC/4
    ADCON0bits.ANS = 0b00; // no analog inputs -> RB0-2 digital
    ADCON0bits.CHS = 0b11; // select 0.6 V reference
    ADCON0bits.ADON = 1; // turn ADC on
    // -> 0.6 V reference ready for sampling

    // configure timer
    OPTION = 0b11010111; // configure Timer0:
    // --0----- timer mode (T0CS = 0) -> RC5 usable
    // ----0--- prescaler assigned to Timer0 (PSA = 0)
    // -----111 prescale = 256 (PS = 111)
    // // -> increment every 256 us
    // // (TMR0<2> cycles every 2.048 ms)
```

```

//*** Main loop
for (;;)
{
    // sample 0.6 V reference
    ADCON0bits.GO = 1;          // start conversion
    while (ADCON0bits.nDONE)    // wait until done
        ;

    // test for low Vdd
    if (ADRES > (int)(0.6/MINVDD*255)) // if measured 0.6 V > threshold
        WARN = 1;                // light warning LED

    // display high nybble for 2.048 ms
    while (!TMR0_2)             // wait for TMR0<2> to go high
        ;
    set7seg(ADRES >> 4);        // output high nybble of result
    TENS_EN = 1;                // enable "tens" digit
    while (TMR0_2)             // wait for TMR0<2> to go low
        ;

    // display low nybble for 2.048 ms
    while (!TMR0_2)           // wait for TMR0<2> to go high
        ;
    set7seg(ADRES & 0x0F);     // output low nybble of result
    ONES_EN = 1;              // enable ones digit
    while (TMR0_2)           // wait for TMR0<2> to go low
        ;
}
}

```

CCS PCB

The initialisation code is much the same as in the previous example, except that we must now select the 0.6 V reference as the ADC input channel, instead of AN2:

```

// configure ADC:
setup_adc(ADC_CLOCK_INTERNAL); // clock = INTOSC/4, turn ADC on
setup_adc_ports(NO_ANALOGS);   // no analog inputs -> RB0-2 digital
set_adc_channel(3);            // select 0.6 V reference
                                // -> 0.6 V reference ready for sampling

```

The main sample and display loop is reused from the previous example, but, again, we need to insert some code to check that VDD is above the minimum allowed value.

The minimum allowable VDD can be defined as:

```
#define MINVDD 3.5 // minimum Vdd (Volts)
```

and the ADC result tested, in a similar way to how it was initially written using XC8, above:

```

// test for low Vdd
if (adc_res > 0.6/MINVDD*255) // if measured 0.6 V > threshold
    output_high(WARN);        // light warning LED

```

Just as in the XC8 example, the use of the floating-point expression '0.6/MINVDD*255' in the comparison causes the compiler to incorporate floating-point routines, making the generated code significantly larger

than it needs to be – 258 words of program memory³, compared with only 135 words for the previous hexadecimal output example.

In the same way as was done with XC8, the unnecessary use of floating-point code can be avoided by casting the expression as an integer:

```
    if (adc_res > (int)(0.6/MINVDD*255))    // if measured 0.6 V > threshold
        output_high(WARN);                //    light warning LED
```

Without the floating-point code, the size of the generated program is reduced to only 145 words of program memory.

Program listing

As in the XC8 version, the only change to the program setup (device configuration, function prototypes etc.) from the previous example is the addition of the following constant definition:

```
/****** CONSTANTS *****/
#define MINVDD 3.5                // minimum Vdd (Volts)
```

And again, most of the rest of the source code is the same as in the previous example, but it is worth listing the main program code, to see the new ADC configuration and how the comparison code fits in:

```
/****** MAIN PROGRAM *****/
void main()
{
    unsigned int8    adc_res;        // result of ADC conversion

    /*** Initialisation

    // configure ports
    setup_comparator(NC_NC_NC_NC);   // disable comps -> RB0-1, RC0-1 digital
    setup_vref(FALSE);              // disable CVref -> RC2 usable

    // configure ADC:
    setup_adc(ADC_CLOCK_INTERNAL);   // clock = INTOSC/4, turn ADC on
    setup_adc_ports(NO_ANALOGS);     // no analog inputs -> RB0-2 digital
    set_adc_channel(3);              // select 0.6 V reference
                                    // -> 0.6 V reference ready for sampling

    // configure Timer0
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_256); // timer mode, prescale = 256
                                                // -> bit 2 cycles every 2.048 ms

    /*** Main loop
    while (TRUE)
    {
        // sample 0.6 V reference
        adc_res = read_adc();

        // test for low Vdd
        if (adc_res > (int)(0.6/MINVDD*255))    // if measured 0.6 V > threshold
            output_high(WARN);                //    light warning LED

        // display high nybble for 2.048 ms
        while (!TMR0_2)                        // wait for TMR0<2> to go high
            ;
        set7seg(adc_res >> 4);                 // output high nybble of result
        output_high(TENS_EN);                 // enable "tens" digit
```

³ using CCS PCB v4.073

```

    while (TMR0_2)                // wait for TMR0<2> to go low
        ;

    // display low nybble for 2.048 ms
    while (!TMR0_2)              // wait for TMR0<2> to go high
        ;
    set7seg(adc_res & 0x0F);      // output low nybble of result
    output_high(ONES_EN);        // enable ones digit
    while (TMR0_2)                // wait for TMR0<2> to go low
        ;
    }
}

```

Comparisons

Here is the resource usage comparison for the “VDD measure” example, including the floating-point and integer arithmetic versions of the C programs:

ADC_Vdd-measure

Assembler / Compiler	Arithmetic	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	integer	104	90	1
XC8 (Free mode)	float	72	508	20
XC8 (Free mode)	integer	72	165	2
CCS PCB	float	67	258	15
CCS PCB	integer	67	145	9

The C source code continues to be significantly shorter than the assembly language version source, and the optimised code generated by the CCS compiler is still more than 50% larger than the assembly version. The real story here, however, is how very inefficient the floating-point versions are, in comparison with integer arithmetic, showing that floating-point operations should be avoided wherever possible.

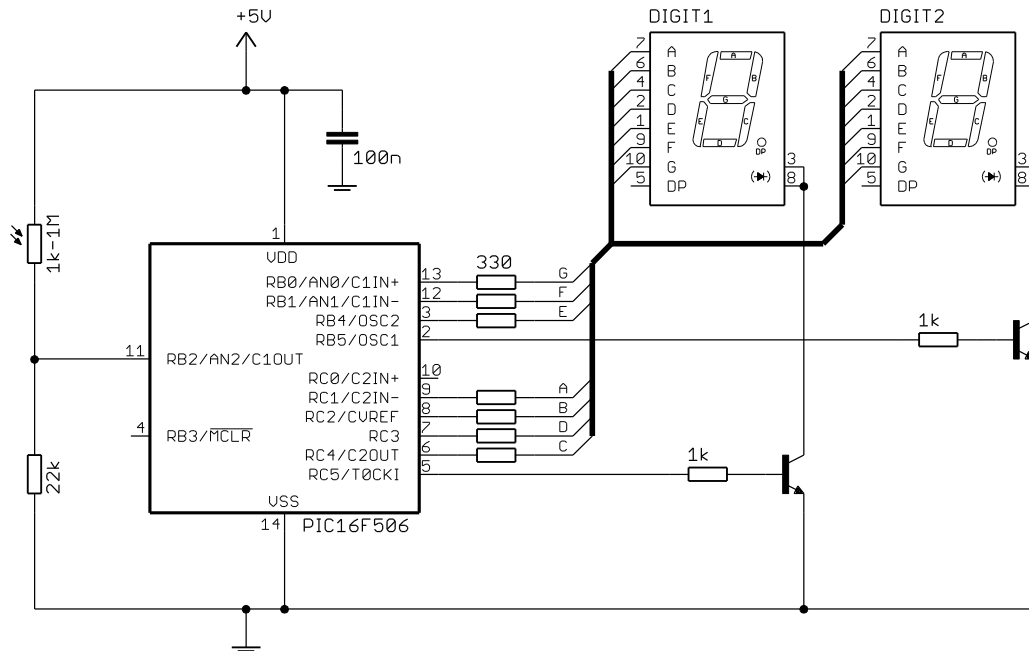
Decimal Output

The light meter presented earlier would be more useful if the light level was represented as a decimal value, instead of hexadecimal. Although we could add a third digit, so that the ADC output between 0 and 255 can be displayed directly in decimal, it would be more meaningful to most people if the result was scaled to a 2-digit result, with the full range being 0 – 99.

The circuit from the hexadecimal output example (shown again on the next page) can be re-used for this. If you are using the [Gooligum baseline training board](#), you should set it up the same way as in that example.

This example was implemented in assembly language in [baseline assembler lesson 11](#), where the main focus of the lesson was on integer arithmetic, including multi-byte addition and subtraction, and 8-bit multiplication. Since the C compiler takes care of the implementing arithmetic operations, we don't need to be concerned with those details here.

To scale the ADC output from 0 – 255 to 0 – 99, it should be multiplied by 99/255. That can be done easily in C, but it is more difficult to do in assembler. In the assembler example, the ADC result was multiplied by



100/256, which is much easier to implement and is only “out” by 0.6%; not really significant, given that the ADC is only accurate to within 0.8%, in any case.

So that the C examples are comparable to the assembler version, we will use the scaling factor of 100/256 here, as well.

XC8

Most of the XC8 program code can be re-used from the hexadecimal output example.

After sampling the analog input, we need to scale the ADC result to 0 – 99, and this scaled result is then referenced twice; once for each digit. So it makes sense to store the scaled result in a variable, which we can declare as:

```
uint8_t    adc_dec;           // scaled ADC output (0-99)
```

because this value will always be small enough (≤ 99) to represent using 8 bits.

To scale the ADC result, we could use:

```
// scale result to 0-99
adc_dec = ADRES * 100/256;
```

However, the XC8 compiler generates smaller code if this is written as:

```
adc_dec = (unsigned)ADRES * 100/256;
```

That is, the 8-bit ADC result in **ADRES** is cast as an unsigned integer.

C compilers usually *promote* smaller integral types (such as ‘char’) to type ‘int’ when they are included in integer arithmetic calculations. In fact, this behaviour is required by the ANSI C standard.

The reason for this “integral promotion” is clear, when we consider how this expression might be evaluated. If the compiler calculates ‘ADRES * 100’ first, it is likely to evaluate to a value greater than 255, which would overflow an 8-bit calculation, leading to incorrect results. Using 16-bit integers to perform these intermediate calculations avoids such problems.

However, C compilers will generally avoid integral promotion in situations where they can conclude that the result will be the same if promotion doesn't occur.

In this case, casting `ADRESH` as an unsigned integer allows the compiler to optimise its code generation, because it can avoid promoting the ADC result to a signed integer and using signed multiplication and division routines; unsigned arithmetic is simpler and therefore requires less code to implement.

Note though that you can't simply assume that a particular change, like this, will make your code smaller – it depends on the specific compiler and its optimisation settings. Sometimes you need to try a number of combinations of type declarations and casting, if you want to generate the smallest possible code.

We then need to extract each digit of the scaled result for display. As we saw in [lesson 5](#), this can be done using the integer division (`/`) and modulus (`%`) operators.

This is best shown in context, within the complete sample and display loop:

```

/***/ Main loop
for (;;)
{
    // sample input
    ADCON0bits.GO = 1;           // start conversion
    while (ADCON0bits.nDONE)    // wait until done
        ;

    // scale result to 0-99
    adc_dec = (unsigned)ADRES * 100/256;

    // display tens digit for 2.048 ms
    while (!TMR0_2)             // wait for TMR0<2> to go high
        ;
    set7seg((unsigned)adc_dec/10); // output tens digit of result
    TENS_EN = 1;                // enable tens digit display
    while (TMR0_2)              // wait for TMR0<2> to go low
        ;

    // display ones digit for 2.048 ms
    while (!TMR0_2)             // wait for TMR0<2> to go high
        ;
    set7seg((unsigned)adc_dec%10); // output ones digit of result
    ONES_EN = 1;                // enable ones digit display
    while (TMR0_2)              // wait for TMR0<2> to go low
        ;
}

```

Again, the `adc_dec` variable has been cast as an unsigned integer in each expression, to optimise code generation.

Finally, because only the decimal digits (0-9) need to be displayed, the additional hexadecimal digits (A-F) can be removed from the lookup tables in the digit display function:

```

/*****/ Display digit on 7-segment display *****/
void set7seg(uint8_t digit)
{
    // pattern table for 7 segment display on port B
    const uint8_t pat7segB[10] = {
        // RB4 = E, RB1:0 = FG
        0b010010, // 0
        0b000000, // 1
        0b010001, // 2

```

```

        0b000001,    // 3
        0b000011,    // 4
        0b000011,    // 5
        0b010011,    // 6
        0b000000,    // 7
        0b010011,    // 8
        0b000011     // 9
    };

    // pattern table for 7 segment display on port C
    const uint8_t pat7segC[10] = {
        // RC4:1 = CDBA
        0b011110,    // 0
        0b010100,    // 1
        0b001110,    // 2
        0b011110,    // 3
        0b010100,    // 4
        0b011010,    // 5
        0b011010,    // 6
        0b010110,    // 7
        0b011110,    // 8
        0b011110     // 9
    };

    // disable displays
    PORTB = 0;           // clear all digit enable lines on PORTB
    PORTC = 0;           // and PORTC

    // output digit pattern
    PORTB = pat7segB[digit]; // lookup and output port B and C patterns
    PORTC = pat7segC[digit];
}

```

CCS PCB

In the CCS version of the hexadecimal example, the result of the ADC conversion was stored in a variable:

```
adc_res = read_adc();
```

Instead of scaling this value and storing the result in another variable, it makes more sense to sample the analog input and scale the result in a single operation, such as:

```
adc_dec = read_adc()*100/256;
```

where the variable, 'adc_dec', has been declared in the same way as 'adc_res' had been:

```
unsigned int8  adc_dec;           // scaled ADC output (0-99)
```

However, you will find that this doesn't work! This code, as written, always sets 'adc_dec' equal to zero.

This happens because the CCS compiler does not perform automatic integral promotion, in the same way that the XC8 compiler does. The 'read_adc()' function returns an 8-bit result, and the expression 'read_adc()*100/256' is evaluated using 8-bit arithmetic operations. Any 8-bit quantity divided by 256 (equivalent to right-shifting it eight times) will always be equal to zero, which is the result we see here.

You might expect that this problem could be overcome by defining 'adc_dec' as a 16-bit 'int16' or 'long' type, but unfortunately that doesn't affect how the expression 'read_adc()*100/256' is evaluated; it is still performed using 8-bit arithmetic, regardless of the type of variable it is assigned to.

The answer is to cast the result of the `read_adc()` function as a 16-bit type:

```
adc_dec = (int16)read_adc()*100/256;
```

This generates the correct result.

This type of problem can be quite difficult to find. You need to be careful in case intermediate values in integer expressions overflow – especially when using the CCS compiler, which, unlike the XC8 compiler, does not automatically promote small integers into larger types.

As in the XC8 version, the digits of the scaled result can be extracted using the integer division (/) and modulus (%) operators.

Again, this is best shown in context, within the complete sample and display loop:

```
// Main loop
while (TRUE)
{
    // sample input and scale to 0-99
    adc_dec = (int16)read_adc()*100/256;

    // display tens digit for 2.048 ms
    while (!TMR0_2) // wait for TMR0<2> to go high
        ;
    set7seg(adc_dec/10); // output tens digit of result
    output_high(TENS_EN); // enable tens digit display
    while (TMR0_2) // wait for TMR0<2> to go low
        ;

    // display ones digit for 2.048 ms
    while (!TMR0_2) // wait for TMR0<2> to go high
        ;
    set7seg(adc_dec%10); // output ones digit of result
    output_high(ONES_EN); // enable ones digit display
    while (TMR0_2) // wait for TMR0<2> to go low
        ;
}
```

And finally, the additional hexadecimal digits (A-F) can be removed from the lookup tables in the digit display function:

```
/***** Display digit on 7-segment display *****/
void set7seg(unsigned int8 digit)
{
    // pattern table for 7 segment display on port B
    const int8 pat7segB[10] = {
        // RB4 = E, RB1:0 = FG
        0b010010, // 0
        0b000000, // 1
        0b010001, // 2
        0b000001, // 3
        0b000011, // 4
        0b000011, // 5
        0b010011, // 6
        0b000000, // 7
        0b010011, // 8
        0b000011 // 9
    };
};
```

```

// pattern table for 7 segment display on port C
const int8 pat7segC[10] = {
    // RC4:1 = CDBA
    0b011110, // 0
    0b010100, // 1
    0b001110, // 2
    0b011110, // 3
    0b010100, // 4
    0b011010, // 5
    0b011010, // 6
    0b010110, // 7
    0b011110, // 8
    0b011110 // 9
};
// disable displays
output_b(0); // clear all digit enable lines on PORTB
output_c(0); // and PORTC

// output digit pattern
output_b(pat7segB[digit]); // lookup and output port B and C patterns
output_c(pat7segC[digit]);
}

```

Comparisons

Here is the resource usage for the “ADC demo with decimal output” assembler and C examples:

ADC_dec-out

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	115	103	7
XC8 (Free mode)	58	423	8
CCS PCB	51	185	15

In this example, where integer arithmetic is involved, the pros and cons of assembler versus C become very apparent. The assembly source is around twice as long as the C versions, reflecting the need to explicitly code the arithmetic operations in assembler. On the other hand, the assembler version generates significantly smaller code – only 56% the size of the optimised CCS version. It is also clear that the XC8 compiler, when running in ‘Free mode’, generates very inefficient code in this example.

Using an Array to Implement a Moving Average

A problem with the decimal-output example above (and the previous hexadecimal-output example) is that that output can become unreadable in flickering light, such as that produced by fluorescent lamps. These flicker at 50 or 60 Hz – too fast for the human eye to notice, but not too quickly for our simple light meter, which samples and displays the changing light level 244 times per second.

As we saw in [baseline assembler lesson 11](#), this problem can be effectively overcome by smoothing, or *filtering*, the raw results before displaying them. Although more advanced (and efficient and effective) filtering algorithms exist, one that is easy to implement is the *simple moving average* (or *box filter*), which averages the last N samples (where N is a fixed number, referred to as the *window size*), giving the same weight to each sample.

To implement this filter, we need to store the last N samples, in an array of size N . Every time a new light level is sampled, the array is updated, with the oldest sample value being overwritten with the new one. Note that it is not necessary to calculate the sum of values in the array every time it is updated; we can instead maintain a running total by subtracting the oldest value and adding the new value to it.

Since the data memory in the PIC16F506 is divided into four banks of 16 registers (plus three shared registers), the largest array that can be allocated as a single object is 16 bytes. That is, we can only easily store the last 16 samples. Since the input is sampled every 4 ms, our filter's window is $16 \times 4 \text{ ms} = 64 \text{ ms}$. This is more than enough to smooth out a 50 Hz flicker, since a 50 Hz signal has a period of only 20 ms.

XC8

To start with, we need to declare the sample array:

```
#define NSAMPLES    16                // size of sample array

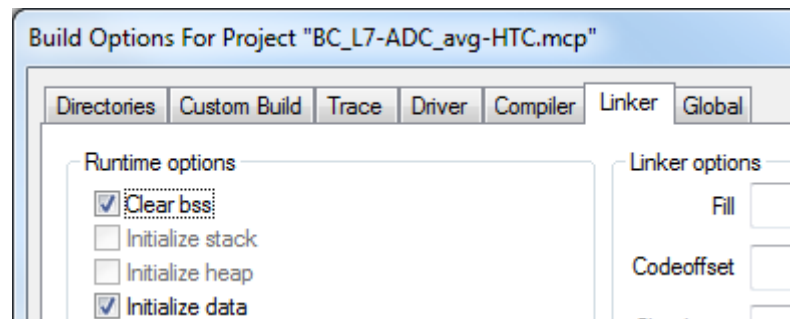
uint8_t smp_buf[NSAMPLES];          // array of samples for moving average
```

Defining the constant, 'NSAMPLES', toward the start of the program, makes it easier to change the number of samples from 16 later, if desired.

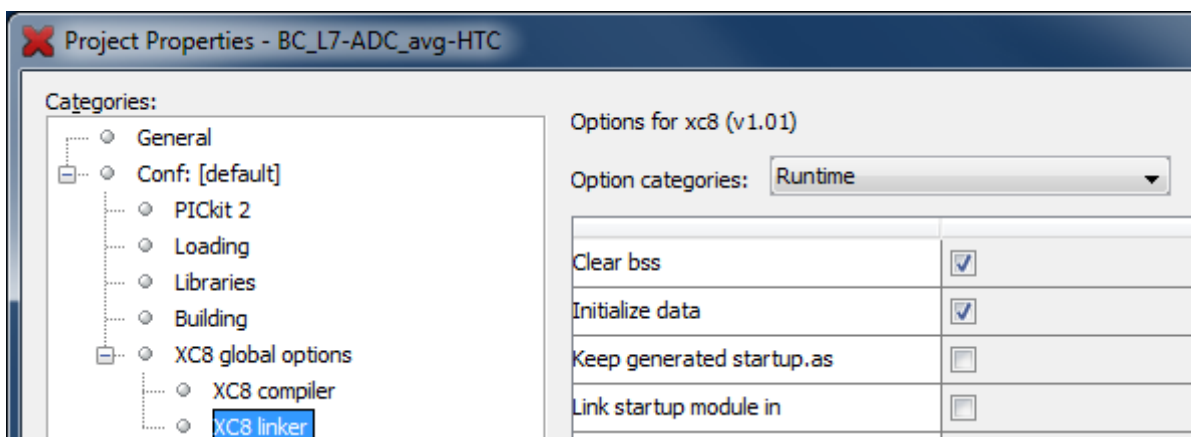
The sample array has to be cleared before it can be used, so that the running total is correct (if the running total is initially zero, the array elements must initially sum to zero; this is easiest to ensure if they are all initially equal to zero). But there is no need to include explicit code to clear the array. All we need to do is to make it a global variable, by declaring it outside any function, including `main()`.

By default, XC8 adds runtime code which, among other things, clears all uninitialized global and static variables, including arrays.

You can check that this option is selected in MPLAB 8 by looking at the "Linker" tab in the project's build options (Project → Build Options... → Project), as shown on the right.



Or, if you are using MPLAB X, you will find the equivalent option within the "Linker" category of the project properties (File → Project Properties, or click on the Project Properties button on the left side of the project dashboard), as shown below:



Whichever version of MPLAB you are using, if the “Clear bss” linker option is selected, the compiler-provided runtime code will clear all the variables.

In addition to the ‘adc_dec’ variable from the last example, we will need variables to store the running total and to keep track of the current sample (used as an index into the sample array):

```
uint16_t    sum = 0;           // running total of ADC samples
uint8_t     adc_dec;         // scaled average (0-99)
uint8_t     s;              // index into sample array
```

The running total (sum) is declared as an unsigned16-bit integer because it needs to be able to hold values up to $16 \times 255 = 4080$, which is too large for an 8-bit variable.

Note that it is zeroed as part of the variable declaration; this saves a line of code later.

The body of the sample and display loop has to be placed within a “for” loop (using ‘s’ as the loop counter), so that each array element is accessed in turn:

```
for (s = 0; s < NSAMPLES; s++)
{
    // sample input
    ...
    // calculate moving average
    ...
    // display digits
}
```

Within the loop, after sampling the input, we update the running total and calculate the average, as follows:

```
// update running total
sum += ADRES - smp_buf[s]; // add new value and subtract old
smp_buf[s] = ADRES;       // update buffer with new value

// calculate average and scale to 0-99
adc_dec = sum / NSAMPLES * 100/256;
```

Complete program

Here is the complete source code for the XC8 version of the “ADC demo with averaged decimal output” program, showing where these code fragments fit in:

```
/*
 * Description: Lesson 7, example 5
 *
 * Displays smoothed ADC output in decimal on 2x7-segment LED displays
 *
 * Continuously samples analog input, averages last 16 samples,
 * scales result to 0 - 99 and displays as 2 x decimal digits
 * on multiplexed 7-seg displays
 *
 *
 * Pin assignments:
 * AN2 = voltage to be measured (e.g. pot or LDR)
 * RB0-1, RB4, RC1-4 = 7-segment display bus (common cathode)
 * RC5 = tens digit enable (active high)
 * RB5 = ones digit enable
 */
```

```

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & IOSCFCS_OFF & OSC_IntrC_RB4EN);

// Pin assignments
#define TENS_EN      PORTCbits.RC5    // tens digit enable
#define ONES_EN     PORTBbits.RB5    // ones digit enable

/***** CONSTANTS *****/
#define NSAMPLES     16              // size of sample array

/***** PROTOTYPES *****/
void set7seg(uint8_t digit);        // display digit on 7-segment display

/***** MACROS *****/
#define TMR0_2      (TMR0 & 1<<2)  // access to TMR0<2>

/***** GLOBAL VARIABLES *****/
uint8_t smp_buf[NSAMPLES];        // array of samples for moving average

/***** MAIN PROGRAM *****/
void main()
{
    uint16_t    sum = 0;              // running total of ADC samples
    uint8_t     adc_dec;             // scaled average (0-99)
    uint8_t     s;                   // index into sample array

    //*** Initialisation

    // configure ports
    TRISB = 0;                       // configure PORTB and PORTC as all outputs
    TRISC = 0;
    CM1CON0 = 0;                      // disable comparator 1 -> RB0, RB1 digital
    CM2CON0 = 0;                      // disable comparator 2 -> RC0, RC1 digital
    VRCON = 0;                       // disable CVref -> RC2 usable

    // configure ADC
    ADCON0bits.ADCS = 0b11;          // clock = INTOSC/4
    ADCON0bits.ANS  = 0b01;          // AN2 (only) analog
    ADCON0bits.CHS  = 0b10;          // select channel AN2
    ADCON0bits.ADON = 1;             // turn ADC on
                                        // -> AN2 ready for sampling

    // configure timer
    OPTION = 0b11010111;             // configure Timer0:
                                        // timer mode (T0CS = 0) -> RC5 usable
                                        // prescaler assigned to Timer0 (PSA = 0)
                                        // prescale = 256 (PS = 111)
                                        // -> increment every 256 us
                                        // (TMR0<2> cycles every 2.048 ms)

    //*** Main loop
    for (;;)

```

```

{
  for (s = 0; s < NSAMPLES; s++)
  {
    // sample input
    ADCON0bits.GO = 1;           // start conversion
    while (ADCON0bits.nDONE)    // wait until done
      ;

    // update running total
    sum += ADRES - smp_buf[s];   // add new value and subtract old
    smp_buf[s] = ADRES;         // update buffer with new value

    // calculate average and scale to 0-99
    adc_dec = sum / NSAMPLES * 100/256;

    // display tens digit for 2.048 ms
    while (!TMR0_2)             // wait for TMR0<2> to go high
      ;
    set7seg((unsigned)adc_dec/10); // output tens digit of result
    TENS_EN = 1;                // enable tens digit display
    while (TMR0_2)             // wait for TMR0<2> to go low
      ;

    // display ones digit for 2.048 ms
    while (!TMR0_2)            // wait for TMR0<2> to go high
      ;
    set7seg((unsigned)adc_dec%10); // output ones digit of result
    ONES_EN = 1;               // enable ones digit display
    while (TMR0_2)            // wait for TMR0<2> to go low
      ;
  }
}

/***** FUNCTIONS *****/

/***** Display digit on 7-segment display *****/
void set7seg(uint8_t digit)
{
  // pattern table for 7 segment display on port B
  const uint8_t pat7segB[10] = {
    // RB4 = E, RB1:0 = FG
    0b010010, // 0
    0b000000, // 1
    0b010001, // 2
    0b000001, // 3
    0b000011, // 4
    0b000011, // 5
    0b010011, // 6
    0b000000, // 7
    0b010011, // 8
    0b000011 // 9
  };

  // pattern table for 7 segment display on port C
  const uint8_t pat7segC[10] = {
    // RC4:1 = CDBA
    0b011110, // 0
    0b010100, // 1
    0b001110, // 2
    0b011110, // 3
  };
}

```

```

        0b010100,    // 4
        0b011010,    // 5
        0b011010,    // 6
        0b010110,    // 7
        0b011110,    // 8
        0b011110    // 9
    };

    // disable displays
    PORTB = 0;        // clear all digit enable lines on PORTB
    PORTC = 0;        // and PORTC

    // output digit pattern
    PORTB = pat7segB[digit];    // lookup and output port B and C patterns
    PORTC = pat7segC[digit];
}

```

CCS PCB

By default, the CCS PCB compiler will only place variables (and arrays) in bank 0.

To instruct the compiler to use the other register banks, place a '#device *=8' directive near the start of the program:

```
#device *=8                // allow variable placement in banks 1-3
```

Once this has been done, variables and arrays can be declared as usual, with the compiler automatically handling their placement.

We can then declare the sample buffer array as:

```
int8  smp_buf[NSAMPLES];    // array of samples for moving average
```

Unlike XC8, the CCS PCB compiler does not automatically clear uninitialized global variables, so it does not matter whether this array is made global or declared within `main()`. Regardless of where it is declared, we need to include a routine, as part of the program initialisation code, to clear the sample array:

```

int8    s;                // index into sample array

// clear sample buffer
for (s = 0; s < NSAMPLES; s++)
    smp_buf[s] = 0;

```

We also need to declare the variables needed for the moving average calculation:

```

int8    adc_res;          // result of ADC conversion
int16   sum = 0;          // running total of ADC samples
int8    adc_dec;          // scaled average (0-99)

```

Note that 'sum' has to be declared as an 'int16' (or 'long'), as this needs to be a 16-bit value. The other variables could be declared as 'char' or 'int', because CCS PCB defines both to be 8-bit types.

As we did in the XC8 example, we need to place the body of the sample and display loop within a "for" loop, to retrieve and update each array element in turn:

```

for (s = 0; s < NSAMPLES; s++)
{
    // sample ADC, calculate moving average, scale and display
}

```

In theory, it should be possible to update the running total and then calculate and scale the moving average as follows:

```
// update running total
sum += (int16)adc_res - smp_buf[s]; // add new value and subtract old
smp_buf[s] = adc_res; // update buffer with new value

// calculate average and scale to 0-99
adc_dec = sum / NSAMPLES * 100/256;
```

Unfortunately, this does not work! **The array is not written to correctly – apparently due to a bug in version 4.073 (and earlier) of the CCS PCB compiler.**

Until CCS releases, and makes freely available, a version of the PCB compiler which corrects this problem, we need to find another way to implement our 16-byte sample buffer.

Luckily, the PCB compiler provides two built-in functions, intended to allow efficient access to registers outside bank 0: ‘read_bank()’ and ‘write_bank()’.

They are most useful in applications where an array would otherwise be used, such as implementing a buffer.

But before using these bank-access functions, we must ensure that the compiler will only use bank 0 by removing the ‘#device * = 8’ directive, so that there is no risk of overwriting registers used by the compiler.

Assuming that we will use bank 1 for the sample buffer, we first have to clear it:

```
// clear sample buffer
for (s = 0; s < NSAMPLES; s++)
    write_bank(1, s, 0);
```

The function ‘write_bank(1, s, 0)’ writes the value ‘0’ to the register at address offset ‘s’ in bank 1, where address offset = 0 is the start of the bank (address 0x30 for bank 1).

The code to update the running total then becomes:

```
// update running total
sum += (int16)adc_res - read_bank(1, s); // add new val and subtract old
write_bank(1, s, adc_res); // update buffer with new value
```

The function ‘read_bank(1, s)’ returns the value in the register at address offset ‘s’ in bank 1.

As you can see, the ‘read_bank()’ and ‘write_bank()’ functions can be substituted quite easily for array reads and writes.

Complete program

Here is the complete source code for the CCS version of the “ADC demo with averaged decimal output” program, using the direct bank-access functions, showing where these code fragments fit within the program:

```
/******
* Description: Lesson 7, example 5b *
* *
* Displays smoothed ADC output in decimal on 2x7-seg LED displays *
* *
* Continuously samples analog input, averages last 16 samples, *
* scales result to 0 - 99 and displays as 2 x decimal digits *
* on multiplexed 7-segment displays. *
******/
```



```

*   Uses bank read and write functions to implement sample buffer   *
*   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *
*****
*   Pin assignments:   *
*   AN2                = voltage to be measured (e.g. pot or LDR)   *
*   RB0-1,RB4,RC1-4    = 7-segment display bus (common cathode)   *
*   RC5                = tens digit enable (active high)           *
*   RB5                = ones digit enable                           *
*   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *
*****/

#include <16F506.h>

/***** CONFIGURATION *****/
// Config: ext reset, no code protect, no watchdog, 4 MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC_IO,IOSC4

// Pin assignments
#define TENS_EN      PIN_C5      // tens digit enable
#define ONES_EN     PIN_B5      // ones digit enable

/***** CONSTANTS *****/
#define NSAMPLES    16          // size of sample buffer

/***** PROTOTYPES *****/
void set7seg(unsigned int8 digit); // display digit on 7-segment display

/***** MACROS *****/
#define TMR0_2      (get_timer0() & 1<<2) // access to TMR0<2>

/***** MAIN PROGRAM *****/
void main()
{
    int8    adc_res;           // result of ADC conversion
    int16   sum = 0;           // running total of ADC samples
    int8    adc_dec;           // scaled average (0-99)
    int8    s;                 // index into sample buffer

    /*** Initialisation

    // configure ports
    setup_comparator(NC_NC_NC_NC); // disable comps -> RB0-1, RC0-1 digital
    setup_vref(FALSE);             // disable CVref -> RC2 usable

    // configure ADC
    setup_adc(ADC_CLOCK_INTERNAL); // clock = INTOSC/4, turn ADC on
    setup_adc_ports(AN2);           // AN2 (only) analog
    set_adc_channel(2);             // select channel AN2
                                    // -> AN2 ready for sampling

    // configure Timer0
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_256); // timer mode, prescale = 256
                                                // -> bit 2 cycles every 2.048 ms

    // clear sample buffer

```

```

for (s = 0; s < NSAMPLES; s++)
    write_bank(1,s,0);

/**** Main loop
while (TRUE)
{
    for (s = 0; s < NSAMPLES; s++)
    {
        // sample input
        adc_res = read_adc();

        // update running total
        sum += (int16)adc_res - read_bank(1,s); // add new, subtract old
        write_bank(1,s,adc_res);             // update buffer with new

        // calculate average and scale to 0-99
        adc_dec = sum / NSAMPLES * 100/256;

        // display tens digit for 2.048 ms
        while (!TMR0_2) // wait for TMR0<2> to go high
            ;
        set7seg(adc_dec/10); // output tens digit of result
        output_high(TENS_EN); // enable tens digit display
        while (TMR0_2) // wait for TMR0<2> to go low
            ;

        // display ones digit for 2.048 ms
        while (!TMR0_2) // wait for TMR0<2> to go high
            ;
        set7seg(adc_dec%10); // output ones digit of result
        output_high(ONES_EN); // enable ones digit display
        while (TMR0_2) // wait for TMR0<2> to go low
            ;
    }
}

/***** FUNCTIONS *****/

/***** Display digit on 7-segment display *****/
void set7seg(unsigned int8 digit)
{
    // pattern table for 7 segment display on port B
    const int8 pat7segB[10] = {
        // RB4 = E, RB1:0 = FG
        0b010010, // 0
        0b000000, // 1
        0b010001, // 2
        0b000001, // 3
        0b000011, // 4
        0b000011, // 5
        0b010011, // 6
        0b000000, // 7
        0b010011, // 8
        0b000011 // 9
    };

    // pattern table for 7 segment display on port C
    const int8 pat7segC[10] = {

```

```

    // RC4:1 = CDBA
    0b011110,    // 0
    0b010100,    // 1
    0b001110,    // 2
    0b011110,    // 3
    0b010100,    // 4
    0b011010,    // 5
    0b011010,    // 6
    0b010110,    // 7
    0b011110,    // 8
    0b011110     // 9
};

// disable displays
output_b(0);           // clear all digit enable lines on PORTB
output_c(0);           // and PORTC

// output digit pattern
output_b(pat7segB[digit]); // lookup and output port B and C patterns
output_c(pat7segC[digit]);
}

```

Comparisons

Here is the resource usage for the “ADC demo with averaged decimal output” assembler and C examples:

ADC_avg

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	150	136	26
XC8 (Free mode)	65	502	27
CCS PCB	61	257	35

In this example, the differences between C and assembly are even more pronounced. The assembly source is more than twice as long as the XC8 and CCS versions, while the assembled version is only around half the size of the optimised code generated by the CCS PCB compiler.

But it’s also clear that, given the problems with compiler bugs and limitations encountered when implementing this example in C, we are hitting the limits of what can be achieved using C compilers on these small baseline devices – something that was not apparent when developing the assembly version.

Summary

The examples in this lesson demonstrate that it is possible to effectively perform analog to digital conversion on baseline PICs, such as the PIC16F506, using either of the XC8 or CCS C compilers. But we have also seen that, although all these compilers make it possible to implement buffers in memory outside bank 0, only the XC8 compiler is able to effectively work directly with “large” (16 byte) arrays.

As expected, source code written for the CCS compiler is consistently the shortest, due to the use of its built-in functions. However, the differences between the CCS and XC8 compilers are dwarfed by that between

assembler and C source, especially for more sophisticated programs, particularly when arithmetic expressions, which can be written succinctly in C, are heavily used:

Source code (lines)

Assembler / Compiler	ADC_hex_out	Vdd_measure	ADC_dec_out	ADC_avg
Microchip MPASM	96	104	115	150
XC8 (Free mode)	68	72	58	65
CCS PCB	63	67	51	61

But again, both C compilers generate code which is significantly larger than the corresponding hand-written assembler versions; the most complex programs being around twice the size of the assembler version, even for the CCS PCB compiler, with “optimised” code generation:

Program memory (words)

Assembler / Compiler	ADC_hex_out	Vdd_measure	ADC_dec_out	ADC_avg
Microchip MPASM	86	90	103	136
XC8 (Free mode)	161	165	423	502
CCS PCB	135	145	185	257

Data memory (bytes)

Assembler / Compiler	ADC_hex_out	Vdd_measure	ADC_dec_out	ADC_avg
Microchip MPASM	1	1	7	26
XC8 (Free mode)	2	2	8	27
CCS PCB	8	9	15	35

There is no doubt that it is much easier to express complex routines in C than assembler, which is reflected in the C code, for all the compilers, being significantly shorter source than the corresponding assembler source code.

On the other hand, it certainly appears that, in the last example, when implementing a “large” sample buffer, we were starting to reach the limit of what can be achieved, with either the CCS or XC8 compilers, on a device as small as the PIC16F506. The CCS PCB compilers had a problem with its implementation of banked array access, suggesting that the baseline PIC architecture just isn’t well suited to the use of C for this type of application. Simple LED flashing and responding to key presses is fine, but when it comes to a moderately sophisticated application, involving analog to digital conversion, with simple digital filtering and scaling, while driving a multiplexed 7-segment display, we appear to have pushed the C compilers nearly as far as they will go. It seems that, to get the most from these baseline PICs, to reach their full potential, we need to use assembler. Or you could pay for the full (optimising) version of XC8, which did not require any workarounds to implement the moving average example, but, with optimisation disabled, generated code which used more than half the memory available on the 16F506.

For anything beyond the simplest applications, instead of trying to fit the solution into the baseline architecture, it often makes more sense to spend a little extra on the microcontroller in order to simplify the programming problem, by moving up to Microchip's "Mid-Range" PIC architecture.

These larger, more flexible microcontrollers are covered in the "[Mid-Range PIC Architecture and Assembly Language](#)" tutorial series, which introduces the mid-range PIC architecture, starting with the PIC12F629. We'll go back to flashing LEDs and responding to pushbutton switches, but we'll see how it can be done, using assembler, on a midrange device.

This is then followed up in the "[Programming Mid-range PICs in C](#)" tutorial series, where we cover the same ground again, using C.

Introduction to PIC Programming

Mid-Range Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 1: Basic Digital Output

The [Baseline PIC Assembler](#) tutorial series introduced the baseline (12-bit) PIC architecture, using devices including the 8-pin digital-only PIC12F509 and 14-pin analog-capable PIC16F506. The series culminated in the development of a simple light meter with smoothed 2-digit decimal output on 7-segment LED displays. However, it was apparent that the limitations of the baseline architecture, such as the lack of interrupts, a maximum of 16 contiguous bytes of banked data memory, and the availability of only a single 8-bit timer, make it difficult to develop applications significantly more complex than this. The baseline architecture's limitations became especially evident when implementing the same examples in C, in the [Baseline PIC C Programming](#) tutorial series.

The mid-range (14-bit) PIC architecture overcomes many of these limitations, offering more memory, larger contiguous blocks of data memory, with simpler and less restricted memory access, more timers, greater flexibility in many areas, additional assembler instructions, a much greater range of peripherals, and support for interrupts – significant, because interrupts allow a different (better) approach to many programming problems, as we will see in later lessons.

This tutorial series introduces the mid-range architecture. Assembly language is used, as that is the best way to gain a thorough understanding of the PIC core and peripherals (languages like C or BASIC hide many of the implementation details, which can make life much easier for the programmer – but the aim here is to gain a good understanding of the underlying hardware).

These lessons assume some familiarity with the content covered in the Baseline PIC Assembler series. Although there is some repetition of material, wherever a topic has been covered in the baseline tutorials, it is described more briefly here, along with a reference to the baseline lesson where the topic was introduced. This approach is practical because the mid-range architecture builds on the baseline architecture we are already familiar with; most of the concepts, and nearly all the assembler instructions, are the same.

This lesson introduces one of the simplest of the mid-range PICs – the PIC12F629. It then goes on to describe basic digital output by lighting and flashing LEDs, as covered in lessons [1](#) and [2](#) of the baseline assembler tutorial series.

In summary, this lesson covers:

- Introduction to the PIC12F629
- Simple digital output to LEDs
- Using loops to create delays
- Using shadow registers to avoid the 'read-modify-write' problem

Getting Started

These tutorials assume that you are using a Microchip PICkit 2 or PICkit 3 programmer and either the [Gooligum Baseline and Mid-range PIC Training and Development Board](#) or Microchip's Low Pin Count Demo Board, with Microchip's MPLAB 8 or MPLAB X integrated development environment. But it is of course possible to adapt these instructions to a different programmers and/or development boards.

See [lesson 0](#) and [baseline lesson 1](#) for more details

As mentioned, we're going to start with one of the simplest of the mid-range PICs – the 8-pin PIC12F629. It is roughly equivalent to the PIC12F509, introduced in [baseline lesson 3](#), but in addition to simple digital I/O, it also includes an analog comparator, a 16-bit timer, and a 128-byte EEPROM. However, it does not include an analog-to-digital converter, nor does it include any advanced peripherals or interfaces. That makes it a good chip to start with; we'll look at the additional features of more advanced mid-range PICs in later lessons.

In summary, for this lesson you should ideally have:

- A PC running Windows (XP, Vista or 7), with a spare USB port
- Microchip's MPLAB 8 IDE software
- A Microchip PICkit 2 or PICkit 3 PIC programmer
- The Gooligum mid-range training board
- A PIC12F629-I/P microcontroller (supplied with the Gooligum training board)

Introducing the PIC12F629

When working with any microcontroller, you should always have on hand the latest version of the manufacturer's data sheet, which, for the 12F629, can be downloaded from www.microchip.com.

The data sheet for the 12F629 also covers the 12F675, which is essentially the same device, with the addition of an analog-to-digital converter (ADC).

The features of various 8-pin PICs are summarised in the following table:

Device	Memory (words or bytes)			Timers		Analog		Clock rate (max MHz)
	Program	Data	EEPROM	8-bit	16-bit	Comp-arators	ADC inputs	
12F508	512	25	0	1	0	0	0	4
12F509	1024	41	0	1	0	0	0	4
12F510	1024	38	0	1	0	1	3	8
12F519	1024	41	64	1	0	0	0	8
12F609	1024	64	0	1	1	1	0	20
12F615	1024	64	0	2	1	1	4	20
12F629	1024	64	128	1	1	1	0	20
12F675	1024	64	128	1	1	1	4	20
12F683	2048	128	256	2	1	1	4	20
12F1501	1024	64	0	2	1	1	4	20
12F1822	2048	128	256	2	1	1	4	32
12F1840	4096	256	256	2	1	1	4	32

The 12F629 has only a little more data memory than the 12F509, but it is arranged differently, as shown in the following register map:

PIC12F629 Registers

Address	Bank 0	Address	Bank 1
00h	INDF	80h	INDF
01h	TMR0	81h	OPTION_REG
02h	PCL	82h	PCL
03h	STATUS	83h	STATUS
04h	FSR	84h	FSR
05h	GPIO	85h	TRISIO
06h		86h	
09h		89h	
0Ah	PCLATH	8Ah	PCLATH
0Bh	INTCON	8Bh	INTCON
0Ch	PIR1	8Ch	PIE1
0Dh		8Dh	
0Eh	TMR1L	8Eh	PCON
0Fh	TMR1H	8Fh	
10h	T1CON	90h	OSCCAL
11h		91h	
		94h	
		95h	WPU
		96h	IOC
		97h	
18h		98h	
19h	CMCON	99h	VRCON
1Ah		9Ah	EEDATA
		9Bh	EEADR
		9Ch	EECON1
		9Dh	EECON2
		9Eh	
1Fh		9Fh	
20h	General Purpose Registers	A0h	Map to Bank 0 20h – 5Fh
5Fh			
60h			
7Fh			
		DFh	
		E0h	
		FFh	

The 12F509’s register map, and the concept of banked register access, was described in [baseline lesson 3](#).

A few differences are immediately apparent:

In the mid-range PICs, each bank consists of 128 registers, compared with only 32 registers in the baseline architecture.

The first 32 addresses in each register bank are used for special function registers (SFRs); the remaining 96 addresses in each bank are available for general-purpose registers (GPRs), allowing much larger contiguous blocks of data memory to be created.

This means that, although the 12F629 has less data memory than the 16F506 (64 bytes compared with 72 bytes), the larger address space of the mid-range architecture means that the 12F629’s 64 bytes are mapped into a single bank, not spread across four banks, as they would be in the baseline architecture.

Note that the GPRs are mapped into both banks, meaning that all data memory in the 12F629 is shared, not banked.

Another significant difference from the baseline architecture is that most SFRs appear in only in one bank or the other. This means that, *when accessing SFRs on mid-range PICs, it is very important to ensure that the correct bank is selected.*

As described in [baseline lesson 3](#), the `banksel` assembler directive will reliably set the bank selection bits for the specified register address. Some SFRs are grouped, so that once the correct bank is selected for one of them, you can be sure that the bank selection will not need to be changed before accessing other registers in the group. But if you are ever in doubt, use `banksel`. And remember that just because two registers happen to be in the same bank in the 12F629, it may not be guaranteed to be true in other mid-range PICs.

Bank selection is controlled by the RP0 bit in the STATUS register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	IRP	RP1	RP0	\overline{TO}	\overline{PD}	Z	DC	C

If $RP0 = 0$, bank 0 is selected; if $RP0 = 1$, bank 1 is selected.

The RP1 and IRP bits are unused on the 12F629; they are used in mid-range devices, such as the 16F690, which have four register banks. In devices with four banks, RP0 and RP1 are used in combination to select the bank for direct register access, while IRP is used select the bank for indirect register access (see [lesson 14](#)) – necessary because FSR, being 8-bits wide, can only point to one of 256 registers, but a four-bank device has 512 register addresses (128 addresses in each bank).

This is much more convenient than the bank selection scheme used in the baseline architecture, where bits in the FSR register were used, which meant that indirect register access could not be done separately from direct register access – a limitation which makes it very difficult for C compilers to implement banked array access on baseline devices, as we saw in [baseline C lesson 7](#). The mid-range architecture has no such limitation.

The remaining bits in the STATUS register, \overline{TO} , \overline{PD} , Z, DC and C, are equivalent to their counterparts in the baseline architecture.

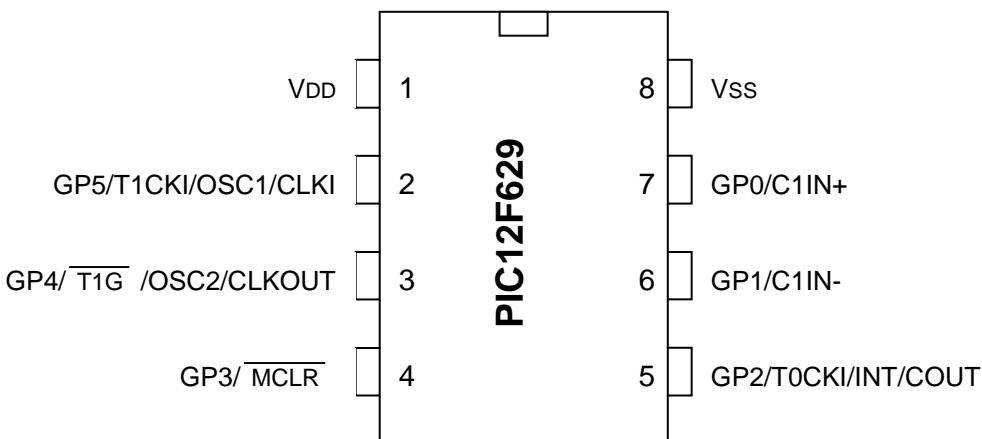
The TRIS (called TRISIO on the 12F629) and OPTION registers are no longer accessed through special instructions, but appear in the register map and are directly accessible, in the same way as any other register. Importantly, this means that these registers are now readable, as well as writable, making it possible to update individual bits.

Note that the OPTION register is called OPTION_REG on mid-range PICs, because “option” is a reserved word in MPASM.

The working register, ‘W’ (equivalent to the ‘accumulator’ in some other microprocessors), is not mapped into memory, and so does not appear in the register map.

PIC12F629 Input/Output

Like the 12F509, the 12F629 provides six I/O pins in an eight-pin package:



VDD is the positive power supply.

VSS is the “negative” supply, or ground. All of the input and output levels are measured relative to VSS.

In most circuits, there is only a single ground reference, at 0 V, and VSS will be connected to ground.

The power supply voltage (VDD, relative to VSS) can range from 2.0 V to 5.5 V, although at least 3.0 V is needed if the clock rate is greater than 4 MHz, and at least 4.5 V is needed to run the PIC at more than 10 MHz.

A *bypass capacitor*, typically 100 nF and preferably ceramic, should be placed between VDD and VSS, as close to the chip as practical, to provide transient power as the current drawn by the PIC changes, and to limit the effect of noise on the power rails. You may find that you can “get away” without using a bypass capacitor, particularly in a small battery-powered circuit. But figuring out why your PIC keeps randomly resetting itself is hard, while 100 nF capacitors are cheap, so include them in your designs!

The remaining pins, GP0 to GP5, are the I/O pins. They are used for digital input and output, except for GP3, which can only be an input. The other pins – GP0, GP1, GP2, GP4 and GP5 – can be individually set to be inputs or outputs.

Note however that each I/O pin has one or more functions that can be assigned to it, such as a comparator output, or a counter input. As we will see later, in some cases these alternate functions need to be disabled before a pin can be used for digital I/O.

Taken together, the six I/O pins comprise the general-purpose I/O *port*, or GPIO port.

If a pin is configured as an output, the output level is set by the corresponding bit in the GPIO register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
GPIO			GP5	GP4	GP3	GP2	GP1	GP0

Setting a bit to ‘1’ outputs a ‘high’ on the corresponding pin; setting it to ‘0’ outputs a ‘low’.

If a pin is configured as an input, the input level is represented by the corresponding bit in the GPIO register. If the input on a pin is high, the corresponding bit reads as ‘1’; if the input pin is low, the corresponding bit reads as ‘0’.

The TRISIO register controls whether a pin is set as an input or output:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TRISIO			TRISIO5	TRISIO4		TRISIO2	TRISIO1	TRISIO0

To configure a pin as an input, set the corresponding bit in the TRISIO register to ‘1’. In the input state, the PIC’s output drivers are effectively disconnected from the pin.

To configure a pin as an output, clear the corresponding TRISIO bit to ‘0’.

By default, each pin is an ‘input’; the TRISIO register is set to all ‘1’s when the PIC is powered on.

Note that TRISIO<3> is greyed-out. Clearing this bit will have no effect because, as mentioned above, the GP3 pin is always an input.

When configured as an output, each I/O pin on the 12F629 can source or sink up to 25 mA – enough to directly drive an LED, without needing an external transistor.

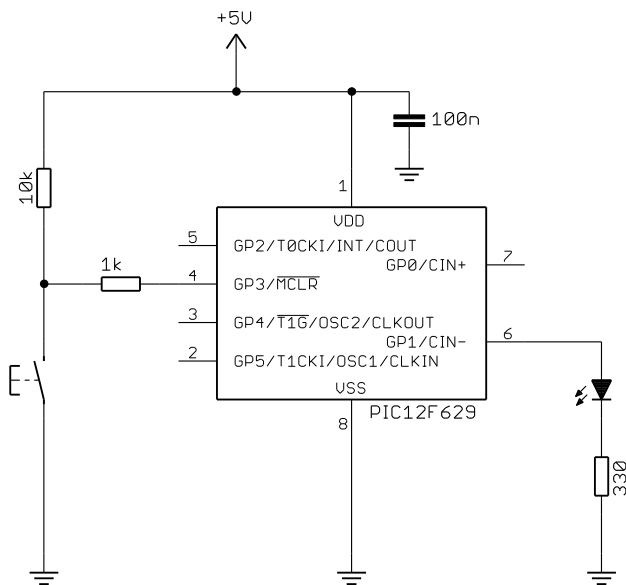
In total, the GPIO port can source or sink up to 125 mA.

Example 1: Turning on an LED

We'll start the same way that we did in [baseline lesson 1](#), by simply lighting a single LED, connected to one of the 12F629's digital I/O pins.

This might appear to be a trivial task, but if you can do something as simple as lighting an LED, you will have proven that you have a functioning circuit, that your PIC code is correct, that you have properly set up an appropriate development environment, and that you can use it effectively to assemble your code and load it into the PIC. When you have achieved all that, you have a firm base to build on.

The complete circuit looks like this:



As you can see, besides the PIC, there isn't very much needed at all.

The power supply should be at least 3 V to properly light the LED. A 5 V supply is assumed in these lessons, reflecting the default voltage provided by the PICkit 2 and PICkit 3 programmers.

A 330 Ω resistor, in series with the LED, is shown here, because that is the value used on the Gooligum training board. But you can choose any value for this resistor, as long as it limits the LED current to no more than 25 mA – the maximum rated current for each pin.

The pushbutton acts as a reset switch.

Pin 4 can be configured as either a digital input (GP3) or as an external reset ("master clear", $\overline{\text{MCLR}}$), which, if pulled low, will reset the processor.

In this example, we'll configure the PIC for external reset. When the pushbutton is pressed, pin 4 will be pulled low, resetting the device. The PICkit 2 and PICkit 3 are also able to pull the reset line low, allowing MPLAB to control $\overline{\text{MCLR}}$ – which is useful for starting and stopping your program.

Of course, when the pushbutton isn't pressed, we want the PIC to run our program, and for that to happen, if external reset is enabled, the $\overline{\text{MCLR}}$ input must be held high. This is what the 10 k Ω pull-up resistor is for; it holds $\overline{\text{MCLR}}$ high while the switch is open¹.

The pushbutton is connected to $\overline{\text{MCLR}}$ via a 1 k Ω resistor. As explained in [baseline lesson 4](#), resistors like this can be used to avoid damage in case an input pin is inadvertently programmed as an output. Such damage is impossible in this case because, as mentioned above, GP3 can only ever be an input. The most important reason for the resistor between pin 4 and the pushbutton is to allow the PIC to be safely and successfully programmed by the PICkit 2 or PICkit 3, using the ICSP programming protocol, when pin 4 is used as the 'VPP' input. During ICSP programming, a high voltage (around 12 V) is applied to VPP, to place the PIC into programming mode. The 1 k Ω resistor is necessary to protect the PICkit 2 or PICkit 3, in case the pushbutton is pressed during programming, grounding the VPP (12 V) signal.

¹ This external pull-up resistor wasn't needed in the baseline PIC examples, because the baseline PICs, and indeed most mid-range PICs, include an internal weak pull-up (see [lesson 3](#)) on $\overline{\text{MCLR}}$ which is automatically enabled whenever the device is configured for external reset.

If you are using the [Gooligum training board](#), plug your PIC12F629 into the top section of the 14-pin IC socket – the section marked ‘12F’². Close jumpers JP3 and JP12 to bring the 10 kΩ resistor into the circuit and to connect the LED to GP1, and ensure that every other jumper is disconnected.

If you have the Microchip Low Pin Count Demo Board, refer back to [baseline lesson 1](#) to see how to build this circuit, either by adding an LED and resistor to the prototyping area or making a connection from GP1 to one of the LEDs on the board via the 14-pin header.

Plug your PICKit 2 or PICKit 3 programmer into the ICSP connector on the training or demo board, with the arrow on the board aligned with the arrow on the PICKit, and plug the PICKit into a USB port on your PC. The PICKit 2 or PICKit 3 can supply enough power for this circuit, so there is no need to connect an external power supply.

With this simple circuit in place, and connected to your PC via a PICKit 2 or PICKit 3 programmer, it’s time to move on to programming!

The [baseline tutorial series](#) explained how to use the MPLAB 8 or MPLAB X environment to create a new assembler project. If you are not familiar with either version of MPLAB, you should follow the instructions in [baseline lesson 1](#), but selecting the 12F629 in the project wizard, instead of the 12F509.

If you choose to use a Microchip-supplied code template, you should choose ‘12F629TMPO.ASM’ in the ‘...\\MPASM Suite\\Template\\Object’ (for MPLAB 8) or ‘...\\mpasmx\\templates\\Object’ (for MPLAB X) directory. But since this template provides a framework for a number of features, including interrupts, which are not covered in this lesson, it is probably best not to include a copy of the template code, but to instead start with an empty file.

If you are using MPLAB 8, after finishing the project wizard, you can create a new (empty) file and add it to your project by selecting the “Project → Add New File to Project...” menu item (also available under the “File” menu, or by right-clicking in the project window), browsing to the project directory, typing a name (ending in ‘.asm’) for the new file, and then clicking “Save”.

Or, if you are using MPLAB X, there are a number of ways to create a new source file and add it to your project, but a simple way is to right-click “Source Files” in the project tree, and select “New → ASM File...”. Enter a name for your new file, select ‘.asm’ as the extension, then click on “Finish”.

To begin writing your program, double-click the assembler source file in the project window. A text editor window will open; it will either be blank, or showing the Microchip-supplied template code (if you created your file from a copy of it), in which case you will need to edit the template code, deleting some parts and changing others, to make it similar to the code presented below.

The MPLAB text editor is aware of PIC assembler (MPASM) syntax and will colour-code text, depending on whether it’s a comment, assembler directive, PIC instruction, program label, etc.

As we did in the [baseline tutorial series](#), we’ll begin each program with a block of comments, giving the name of the program, modification date and version, who wrote it, and a general description of what it does. The template code includes a “Files required” section. This is useful in larger projects, where your code may rely on other modules; you can list any dependencies here. We’ll also document what processor

² Note that, although the PIC12F629 comes in an 8-pin package, **it will not work** in the 8-pin ‘10F’ socket. You must install it in the ‘12F’ section of the 14-pin socket.

the code is written for, and how each pin is used – and anything else which will help anyone working on this code that needs to understand what the program does, and how.

MPASM comments begin with a ‘;’. They can start anywhere on a line. Anything after a ‘;’ is ignored by the assembler.

For example:

```
;*****
;
;  Filename:      MA_L1-Turn_on_LED.asm
;  Date:         1/5/12
;  File Version: 1.2
;
;  Author:       David Meiklejohn
;  Company:     Gooligum Electronics
;
;*****
;
;  Architecture: Mid-range PIC
;  Processor:    12F629
;
;*****
;
;  Files required: none
;
;*****
;
;  Description:   Lesson 1, example 1
;
;  Turns on LED. LED remains on until power is removed.
;
;*****
;
;  Pin assignments:
;      GP1 = indicator LED
;
;*****
```

Next we need to tell MPLAB what processor we’re using:

```
list      p=12F629
#include   <p12F629.inc>
```

The first line tells the assembler which processor to assemble for. It’s not strictly necessary, as it is set in MPLAB (configured when you selected the device in the project wizard). MPLAB displays the processor it’s configured for at the bottom of the IDE window; see the screen shot above. Nevertheless, you should always use the `list` directive at the start of your assembler source file, in case you have accidentally selected the wrong processor in MPLAB. If there is a mismatch between the `list` directive and MPLAB’s setting, MPASM will warn you and you can correct the problem.

The next line uses the `#include` directive which causes an *include file* (`p12F629.inc`, located in the ‘...\\MPASM Suite’ directory) to be read by the assembler. This file sets up aliases, or *labels*, for all the features of the 12F629, so that we can refer to registers etc. by name (e.g. ‘GPIO’) instead of numbers, as was explained in [baseline lesson 6](#).

So, to correctly specify which processor (such as 12F629) is to be used, you need to select that processor when you set up the project in MPLAB and include appropriate `list` and `include` directives in the assembler source.

Next the processor is configured:

```
__CONFIG    _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
           _PWRTE_ON & _INTRC_OSC_NOCLKOUT
```

[this directive must be written as a single line in the assembler source code]

Mid-range PICs have one or more “configuration words” (sometimes referred to as *fuses*), mapped outside the user program memory space, which define a number of aspects of the processor’s configuration. Like the baseline PICs, the 12F629 has a single configuration word.

The `__CONFIG` directive is used to define the value(s) to be loaded into the configuration word(s). It is usually used with labels (defined in the processor’s include file) representing values which are intended to be ANDed together to set or clear the configuration bits corresponding to the options being selected. We’ll examine these in greater detail in later lessons, but briefly the options being selected here are:

- `_MCLRE_ON`

Enables the external reset, or “master clear” ($\overline{\text{MCLR}}$) on pin 4.

As mentioned above, if external reset is enabled, pulling this pin low will reset the processor. Or, if external reset is disabled, the pin can be used as an input: **GP3**.

Unless you need to use every pin for I/O, it’s a good idea to enable external reset by including ‘`_MCLRE_ON`’ in the `__CONFIG` directive.

- `_CP_OFF`

Turns off program memory code protection.

When your code is in production and you’re selling PIC-based products, you may want to prevent others (such as competitors) from accessing your code. If you specify `_CP_ON`, the program memory will be protected, meaning that if someone tries to use a PIC programmer to read it, all they will see are zeros.

- `_CPD_OFF`

Turns off data memory code protection.

The 12F629 includes “EEPROM” (more correctly, “flash”) data memory, which is separate to the register file address space, and is accessed indirectly through special function registers. This memory is non-volatile; it retains its contents when the PIC is powered off. EEPROM data may be considered to be an integral part of the program, and worthy of protection. If you specify `_CPD_ON`, the EEPROM memory will be protected; its contents cannot be accessed by an external PIC programmer. Or the EEPROM may be used to hold data, such as system configuration or logged data, which the user should be able to access, even if the program code is protected. To provide this flexibility, program and data (EEPROM) memory are protected independently.

- `_BODEN_OFF`

Disables brown-out detection.

The PIC’s operation can become unreliable if the supply voltage drops too low, which can happen during a *brown-out*, when the supply voltage sags, but does not fall quickly to zero. The 12F629 has brown-out detect circuitry, which will reset the PIC in a brown-out situation, if `_BODEN_ON` is selected. But if your power supply is not likely to suffer from brown-outs, you can leave this feature disabled.

- `_WDT_OFF`

Disables the watchdog timer.

As we saw in [baseline lesson 7](#), the watchdog timer provides a means of automatically restarting a crashed program, or to regularly wake the device from sleep. Although the watchdog timer is very useful in a production environment, it can be a nuisance when prototyping, so it is best left disabled to begin with.

- `_PWRTE_ON`

Enables the power-up timer.

When a power supply is first turned on, it can take a while for the supply voltage to stabilise, during which time the PIC's operation may be unreliable. If the power-up timer is enabled, the PIC is held in reset (it does not begin running the user program) for some time, nominally 72 ms, after the supply voltage reaches a minimum level.

For reliable operation, you should leave this option enabled, unless you are using an external supervisor circuit, which monitors system voltages and controls the PIC's external reset.

- `_INTRC_OSC_NOCLKOUT`

Selects the internal RC oscillator as the clock source, with no clock output.

PICs can be clocked in a number of ways, as we saw for the 12F509 in [baseline lesson 7](#) and the 16F506 in [baseline lesson 8](#). The 12F629 supports the same clock options as the 16F506, although without the ability to select the frequency of the internal 'RC' oscillator, which on the 12F629 always runs at a nominal 4 MHz. It is not as accurate or stable as an external crystal, but has the advantage of not needing any external components and leaves all of the PIC's pins free for I/O, unless the instruction clock (one quarter of the processor clock rate, i.e. 1 MHz, or 1 μ s per instruction, given a 4 MHz processor clock) is output on CLKOUT.

To turn on an LED, we don't need accurate timing. And there is no need to make the clock signal available externally, so the `_INTRC_OSC_NOCLKOUT` option is appropriate for this application.

If you have based your project on the Microchip-supplied template code, you will see that the next sections in the template relate to defining variables and initialising the EEPROM with data. Since we do not need to use variables or the EEPROM in this example, you can safely delete these sections.

The next section of the template code refers to the oscillator calibration value:

```

;-----
; OSCILLATOR CALIBRATION VALUE
;-----

OSC          CODE          0x03FF

```

The `CODE` directive is used to introduce a *section* of program code.

The `0x03FF` after `CODE` is an address in hexadecimal (signified in MPASM by the '0x' prefix). Program memory on the 12F629 extends from 0000h to 03FFh. This `CODE` directive is telling the linker to place the section of code that follows it at 0x3FF – the very top of the 12F629's program memory.

However, in this case, there *is* no code following this first `CODE` directive. Instead, this is simply a marker to remind us that the oscillator calibration value is held, as an instruction, at the top of program memory.

Like the 12F509, the speed of the internal RC oscillator in the 12F629 can be varied over a small range by changing the value of the `OSCCAL` register, to compensate for variability in the manufacturing process. Microchip tests every 12F629 in the factory, and calculates the value which, if loaded into `OSCCAL`, will make the oscillator run as close as possible to 4 MHz. This calibration value is inserted into the instruction placed at the top of the program memory (0x3FF), which is:

```
retlw k
```

where 'k' is the calibration value inserted in the factory.

A value like this, which is embedded in an instruction, is referred to as a *literal*.

As explained in [baseline lesson 3](#), the `retlw` instruction is used to exit a subroutine, returning a value in the `W` register to the code which called the subroutine – “return with literal in **W**”.

This is different from the scheme used in the baseline architecture, where the instruction at the top of program memory, which loads the calibration value into `W`, is the first instruction executed when the PIC is reset, and program execution “wraps around” at the start of memory.

Instead, in the mid-range architecture, the *reset vector*, where program execution begins, is always at the start of memory: address 0000h.

On a PIC12F629, the user program, beginning at 0000h, can choose to *call* the calibration “subroutine” (consisting of a single `retlw` instruction, as above) at the end of program memory, to “look up” the correct oscillator calibration for this device. Or, if the internal RC oscillator is not being used, or if exact timing is not important, the calibration instruction can simply be ignored.

In the baseline examples, we used the `res` directive in a construct like this:

```
RESET   CODE    0x3FF          ; processor reset vector
        res     1             ; holds internal RC cal value, as a movlw k
```

to reserve the program memory used by the calibration instruction, ensuring that it could not be overwritten by the user program. This is not necessary when using the default, Microchip-supplied linker script for the 12F629, because that script (unlike the ones that Microchip supply for the baseline PICs) declares the memory used by the calibration instruction to be “protected”, so that it will not be overwritten.

Therefore, there is no need to include a `CODE` directive, like either of those above, in our program. It is only useful for documentation, but that is not really necessary, since we can adequately comment the code which loads `OSCCAL` – see below. But of course, how you choose to comment your code is very much a matter of personal style.

The next sections in the Microchip-supplied template consist of code used to implement an *interrupt service routine (ISR)* (to be introduced in [lesson 6](#)) and some code to jump around the ISR. Since we are not using interrupts in this example, these sections can be deleted.

Since we are using the internal RC oscillator, we should start the program by calibrating it:

```
RESET   CODE    0x0000        ; processor reset vector
        ; calibrate internal RC oscillator
        call    0x03FF        ; retrieve factory calibration value
        banksel OSCCAL        ; (stored at 0x3FF as a retlw k)
        movwf  OSCCAL        ; then update OSCCAL
```

Because program execution begins at address 0x0000, this address is specified in the `CODE` directive, placing this code section at the start of program memory, so that it will be executed whenever the PIC is powered on or reset. Another way to say this is that the *program counter*, which points to the next instruction to be executed, is initialised to 0x0000 when the PIC is reset.

This code section is labelled `RESET` here, but you can use any label you want, as long as it’s not a reserved word and is not the name of any other code section in your program.

Next the oscillator calibration value is retrieved, by using the `call` instruction (“**call** subroutine”) to call the calibration instruction at the end of program memory, which returns with the factory calibration value in `W`, as described above. Note again that this scheme is different from that used in the baseline devices.

The calibration value can then be written to the `OSCCAL` register, but before doing so, the bank selection bits must be configured to allow it to be accessed. As mentioned above, this is an important difference

between the baseline and mid-range architectures. On mid-range devices, such as the 12F629, you must ensure that the correct bank is selected when accessing special function registers. The best way to ensure this, avoiding errors and making your code more portable, is to use the ‘banksel’ directive, as shown in the code above, and as explained in [baseline lesson 3](#).

Finally, the ‘movwf’ instruction – “**move W to file register**” – is used to copy (“move”, in Microchip-speak) the factory calibration value, held in W, into the OSCCAL register.

At this point, all the preliminaries are out of the way. The processor has been specified, the configuration set, and the oscillator calibration value updated.

Next it is usual to initialise special function registers, to configure the PIC’s ports and peripherals appropriately.

In this case, we need to configure the GP1 pin as an output:

```

; configure port
movlw  ~ (1<<GP1)          ; configure GP1 (only) as an output
banksel TRISIO
movwf  TRISIO

```

Recall that, to configure a pin as an output, the corresponding bit in the TRISIO register must be cleared; by default the TRIS bits are set to ‘1’, meaning that all pins are configured as inputs at power-up.

The first instruction, ‘movlw’ – “**move literal to W**” – loads a value into W.

We could have written this instruction as:

```

movlw  b'111101'          ; configure GP1 (only) as an output

```

Note that to specify a binary number in MPASM, the syntax b’*binary digits*’ is used, as shown.

This binary value, when loaded into TRISIO, will configure GP1 as an output, leaving the remaining pins configured as inputs.

However, it is often clearer to make use of expressions containing symbols defined in the processor include file, such as ‘GP1’, instead of writing binary constants. For example, the expression ‘~ (1<<GP1)’ is equivalent to the binary constant b’1111101’ (only six bits of this value need be specified in the instruction above, because the top two bits of TRISIO are unused). Another advantage of using symbols is that mistyping a symbol is likely to be picked up by the assembler, while mistyping a binary constant is likely to be missed, making the use of symbols less error-prone.

Having loaded the correct value into W, the ‘movwf’ instruction is used to write it to TRISIO. And, of course, banksel is used to select the bank containing TRISIO, before it is accessed.

*Note: The tris instruction is **not** used to write to the TRIS registers on mid-range devices. The TRIS registers are accessed using general instructions, such as movwf.*

To make GP1 output a ‘high’, we have to set bit 1 of GPIO to ‘1’.

This could be done by:

```

banksel GPIO
movlw  1<<GP1            ; set GP1 high
movwf  GPIO

```

using the ‘movlw’ and ‘movwf’ instructions we have already seen.

The remaining bits in GPIO are cleared, but since the other pins are all inputs, it doesn't matter, in this example, what their corresponding GPIO bits are set to.

However, in many cases you will want to set or clear a single bit, while leaving the other bits in a register unchanged. This can be done with the bit set and clear instructions:

'bsf f,b' sets bit 'b' in register 'f' to '1' – "bit set file register".

'bcf f,b' clears bit 'b' in register 'f' to '0' – "bit clear file register".

These instructions, and any like them, which operate by reading a register, modifying its contents, and then writing the changed value back to the register, can create problems when used with port registers, such as GPIO. This is referred to as the *read-modify-write* problem, and is explained in more detail in [baseline lesson 2](#). It can happen because, in the mid-range and baseline architectures, whenever an instruction reads a port register, the external pins are read, not the internal "output latch" which had been written to. This means that, if an output is slow to change because of a capacitive load, or is being held low or high by an excessive external load, the value read may not match the value written to it. And that can lead to unexpected results, when using instructions such as 'bsf' and 'bcf'.

However, in this simple example, it is very unlikely that there will be any problem with simply turning on a single output, since we are not making any fast changes (and hence capacitive loading is not an issue), we are not changing multiple pins in the same port using sequential instructions (not giving a pin time to change, before being read by the next instruction) and there is no significant load on the pin. So it is safe to use:

```
banksel GPIO
bsf     GPIO,GP1      ; set GP1 high
```

If we leave it there, when the program gets to the end of this code, it will continue executing whatever instructions happen to be in the rest of the program memory; not what we want! So we need to get the PIC to just sit doing nothing, with the LED still turned on, until it is powered off.

What we need is an "infinite loop", where the program does nothing but loop back on itself, indefinitely. Such a loop could be written as:

```
here    goto    here
```

'here' is a label representing the address of the goto instruction.

'goto' is an unconditional branch instruction. It tells the PIC to **go to** a specified program address.

This code will simply go back to itself, always. It's an infinite, do-nothing, loop.

A shorthand way of writing the same thing, that doesn't need a unique label, is:

```
goto    $          ; loop forever
```

'\$' is an assembler symbol meaning the current program address.

So this line will always loop back on itself.

Finally, at the end of your program source, you must include an 'END' directive.

If you put together all the pieces of code presented above, and assemble it, the assembler will give you a couple of messages like:

```
Message[302] C:\...\MA_L1-TURN_ON_LED.ASM 49 : Register in operand not in bank 0. Ensure that bank bits are correct.
```

These messages are generated whenever your code references a register which is not in bank 0, to remind you that you should be taking care to set the bank selection bits correctly. Since we have been taking care to ensure that the bank selection bits are correct, it can be annoying to see these messages – particularly in a larger program, where there will be many more of them. And worse, having a large number of unnecessary messages can make it easy to miss more important messages and warnings.

Luckily, messages and warnings can be disabled, using the ‘errorlevel’ directive:

```
errorlevel -302 ; no warnings about registers not in bank 0
```

This should be placed toward the beginning of your program.

Complete program

Putting together all the above, here’s our complete assembler source for turning on an LED:

```

;*****
;
; Description: Lesson 1, example 1
;
; Turns on LED. LED remains on until power is removed.
;
;*****
;
; Pin assignments:
; GP1 = indicator LED
;
;*****

list p=12F629
#include <p12F629.inc>

errorlevel -302 ; no warnings about registers not in bank 0

;***** CONFIGURATION
; ext reset, no code or data protect, no brownout detect,
; no watchdog, power-up timer, 4 Mhz int clock
__CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

;***** RESET VECTOR *****
RESET CODE 0x0000 ; processor reset vector
; calibrate internal RC oscillator
call 0x03FF ; retrieve factory calibration value
banksel OSCCAL ; (stored at 0x3FF as a retlw k)
movwf OSCCAL ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start

```

```

; configure port
movlw  ~ (1<<GP1)      ; configure GP1 (only) as an output
banksel TRISIO
movwf  TRISIO

;***** Main code
; turn on LED
banksel GPIO
bsf    GPIO,GP1      ; set GP1 high

; loop forever
goto  $

END

```

Now that you have the complete assembler source, you can build the application, which involves assembling the source files to create object files, and then linking the object files to build the executable code. Normally this is transparent; MPLAB does both steps for you in a single operation. It is really only important to know that assemble and link steps are separate operations when working with projects that consist of multiple source files or libraries of pre-assembled routines.

The build process is shown in detail in [baseline lesson 1](#), but, briefly, to build a project in MPLAB 8, select the “Project → Make” menu item, press F10, or click on the “Make” toolbar button:



“Make” will assemble any source files which need assembling (i.e. ones which have changed since the last time the project was built), then link them together.

If you are using MPLAB X, you should first ensure that your project is the “main” project – it should be highlighted in bold in the Projects window. If not, right-click it and select “Set as Main Project”.

To build the project, right-click it in the Projects window and select “Build”, or select the “Run → Build Main Project” menu item, or simply click on the “Build Main Project” button (looks like a hammer) in the toolbar:



This will assemble any source files which have changed since the project was last built, and link them.

The final step is to load (program) the final assembled and linked code into the PIC. This process is also shown in more detail in [baseline lesson 1](#).

If you are using a PICkit 2 or PICkit 3 programmer, the PIC12F629 can be programmed from within MPLAB 8 or MPLAB X.

In MPLAB 8, select PICkit 2 or PICkit 3 from the “Programmer → Select Programmer” submenu.

If you are using a PICkit 3, you may see messages telling you that new firmware must be downloaded, or warning you that the voltage may be too high – just click ‘OK’ on these. You also need to tell your PICkit 3 to provide power. Open the PICkit 3 Settings window by selecting the “Programmer → Settings” menu item and then in the “Power” tab, select “Power target circuit from PICkit 3”.

After you select your programmer, an additional toolbar will appear.

For the PICKit 2, it looks like: 


For the PICKit 3, we have: 

The first icon (on the left) is used to initiate programming. When you click on it, you should see messages telling you that the PIC is being programmed and verified.

Your PIC should now be programmed!

If you are using a PICKit 3, the LED on GP1 should immediately light.

If you have a PICKit 2, you won't see anything yet. That is because, by default, the PICKit 2 holds the $\overline{\text{MCLR}}$ line low after programming. Since we have used the `_MCLRE_ON` option, enabling external reset, the PIC is held in reset and the program will not run. If the external reset was disabled, the LED would have lit as soon as the PIC was programmed.


To allow the program to run, click on the  icon.

The LED should now light up!

If you are using MPLAB X, you must first ensure that your PICKit 2 or PICKit 3 is selected as the hardware (programmer) tool in the project properties window, which you can open by right-clicking your project in the Projects window and selecting "Properties", or simply click on the "Project Properties" button on the left side of the Project Dashboard.

While in the project properties window, if you have a PICKit 3, you should ensure that the "Power target circuit from PICKit3" option, under the PICKit 3's "Power" category, is selected.

To program the PIC and run your program (in a single operation):


- Right-click your project in the Projects window, and select "Run", or
- Select the "Run → Run Main Project" menu item, or
- Press 'F6', or
- Click on the "Make and Program Device" button in the toolbar: 


Whichever of these you choose, you should see output messages ending in:

Running target...

The LED on GP1 should now light.

Being able to build, program and run in a single step, by simply pressing 'F6' or clicking on the "Make and Program Device" button is very useful, but what if you don't want to automatically run your code, immediately after programming?

If you want to avoid running your code, click on the "Hold in Reset" toolbar button  before programming. You can now program your PIC as above.

Your code won't run until you click the reset toolbar button again, which now looks like  and is now tagged as "Release from Reset".

Example 2: Flashing an LED (50% duty cycle)

Having lit a single LED, the next step is to make it flash.

Although it is often preferable to make use of timer-driven interrupt routines (as we will see in [lesson 6](#)) to do something like flashing an LED in the “background”, the simplest approach is to simply light the LED, wait for some time by using a fixed delay, toggle the LED, wait again, and then repeat.

Or, if the LED is going to be on half the time (on for the same period that it is off, for a 50% *duty cycle*), we can simply continue to repeatedly toggle the LED, following a single fixed delay, as expressed in the following pseudo-code:

```
start with LED off
repeat
    delay 500 ms
    toggle LED
done
```

Note that the 500 ms delay gives a total flash period of 1 s, meaning that the LED is flashing at 1 Hz.

But first, you’ll need to create a new project. It makes sense to base it on the project and code you created in example 1; one method for doing this is given in [baseline lesson 2](#).

The configuration sections of the code (specifying the device and its configuration) remain the same, but of course you should update the comments to reflect this new project.

If you want really accurate timing, you’d use a crystal or external clock source, but the internal RC oscillator is good enough for simple LED flashing. Nevertheless, to make the LED flash timing as accurate as possible, it’s important to include the oscillator calibration code at the start of your program.

To generate the delay, we need to make the PIC “do nothing” for some amount of time, and, as explained in more detail in [baseline lesson 2](#), this means implementing *delay loops*.

A loop needs a loop counter: a variable which is incremented or decremented on every pass through the loop.

Variables are defined by reserving data memory (or general purpose registers), using the ‘UDATA’ and ‘res’ directives. For example:

```
;***** VARIABLE DEFINITIONS
        UDATA
dc1     res 1           ; delay loop counters
dc2     res 1
```

However, if you include these directives in your program for the PIC12F629, you will find that, although the code compiles ok, the build fails in the link phase, with an error like:

```
Error - section '.udata' can not fit the section.
```

What’s going on?

[Baseline lesson 3](#) explained that, on many baseline PICs, some registers are *banked*, being mapped into only one of the PIC’s banks of data memory, while another set of registers (usually much smaller) are *shared*, or unbanked, being mapped into every bank. This is also true for mid-range PICs.

The ‘UDATA’ directive declares a section of banked data memory.

If you do not specify a label for a UDATA section, MPASM will name it ‘.udata’.

Recall that the 12F629 does not have any banked data memory; it is all shared. So this error message is telling us that the linker cannot find space for our UDATA section, because there is no banked memory to put it into.

Therefore, on the 12F629 (or any mid-range PIC without banked GPRs), all variables must be defined using 'UDATA_SHR', which declares a section of shared data memory, instead of 'UDATA'.

For example:

```
;***** VARIABLE DEFINITIONS
        UDATA_SHR
dc1     res 1           ; delay loop counters
dc2     res 1
```

This will declare two single-byte variables, 'dc1' and 'dc2', in shared memory.

And note that, since the variables are held in shared memory, there is no need to use `banksel` before accessing them.

Here's an example of a simple "do nothing" delay loop:

```
        movlw    .N
        movwf    dc1           ; dc1 = 10 = number of loop iterations
dly1    nop
        decfsz   dc1,f
        goto     dly1
```

The first two instructions initialise the loop counter variable 'dc1' to the decimal value "N". Since the mid-range PICs are 8-bit devices, "N" has to be between 0 and 255.

Note that numbers in MPASM are specified as being decimal constants by prefixing them with a '.', or using the syntax d'*decimal digits*'. If you don't do this, the assembler will use the default *radix* (hexadecimal), and you may not be using the number you think you are! Although it's possible to set the default radix to decimal, you'll run into problems if you rely on a particular default radix being set, and then later copy and paste your code into another project, with a different default radix, giving different results. It's much safer to simply prefix all decimal numbers with '.'.

The 'decfsz' instruction performs the work of implementing the loop – “**d**ecre**me**nt **f**ile register, **s**kip if **z**ero”. First, it decrements the contents of the specified register, and either writes the result back to the file register (if 'f' is specified as the destination) or to W, (if 'w' is specified as the destination). If the result is not yet zero, the next instruction is executed, which will normally be a 'goto' which jumps back to the start of the loop. But if the result is zero, the next instruction is skipped, exiting the loop.

Mid-range PICs also have an 'incfsz' instruction, equivalent to 'decfsz', except that it increments a file register instead of decrementing it. It's used in loops where you want to count up from an initial value, instead of down.

For a 'decfsz' loop, the number of loop iterations is equal to the initial value of the loop counter ("N" in the example above), assuming it is greater than zero.

The 'nop' instruction – “**n**o **o**peration” – was included to pad out the example delay loop, to make the delay longer. It does nothing but take some time to execute.

How much time depends on the clock rate. Instructions are executed at one quarter the rate of the processor clock. In this case, the PIC is using the internal RC clock, running at a nominal 4 MHz. The instructions are clocked at ¼ of this rate: 1 MHz. So in this example, each instruction cycle is 1 µs.

Most mid-range PIC instructions, including 'nop', execute in a single cycle. The exceptions are those which jump to another location, such as 'goto', which take two cycles to execute.

This means that another useful "do nothing" instruction is 'goto \$+1'. Since '\$' stands for the current address, '\$+1' is the address of the next instruction. Hence, 'goto \$+1' jumps to the following instruction – apparently useless behaviour. But like all 'goto' instructions, it executes in two cycles. So 'goto \$+1' provides a two cycle delay in a single instruction – equivalent to two 'nop's, but using less program memory.

The ‘`decfsz`’ instruction normally executes in a single cycle. But if the result is zero, and the next instruction is skipped, an extra cycle is added, making it a two-cycle instruction.

To calculate the total time taken by the loop, add the execution time of each instruction in the loop:

```

nop                1
decfsz  dc1, f     1 (except when result is zero)
goto    dly1      2

```

That’s a total of 4 cycles, except the last time through the loop, when the `decfsz` takes an extra cycle and the `goto` is not executed (saving 2 cycles), meaning the last loop iteration is 1 cycle shorter. And there are two instructions before the loop starts, adding 2 cycles.

Therefore the total delay time = $(N \times 4 - 1 + 2)$ cycles = $(N \times 4 + 1)$ μ s

If there was no ‘`nop`’, the delay would be $(N \times 3 + 1)$ μ s.

It may seem that, because 255 is the highest 8-bit number, the maximum number of iterations (N) should be 255. But not quite. If the loop counter is initially 0, then the first time through the loop, the ‘`decfsz`’ instruction will decrement it to 255, which is non-zero, and the loop continues – another 255 times.

Therefore the maximum number of iterations is in fact 256, with the loop counter initially 0.

So for the longest possible single loop delay, we can do something like:

```

                clr f    dc1                ; loop 256 times
dly1  nop
                decfsz  dc1, f
                goto    dly1

```

The two “move” instructions have been replaced with a single ‘`clrf`’ instruction, which clears (to 0) the specified register – “clear file register”.

This uses 1 cycle less, so the total time taken is $256 \times 4 = 1024$ μ s \approx 1 ms.

That’s still well short of the 0.5 s needed, so we need to wrap (or *nest*) this loop inside another, using separate counters for the inner and outer loops, as shown:

```

                movlw   .N                ; loop (outer) N times
                movwf  dc2
                clr f    dc1                ; loop (inner) 256 times
dly1  nop                    ; inner loop = 256 x 4 - 1 = 1023 cycles
                decfsz  dc1, f
                goto    dly1
                decfsz  dc2, f
                goto    dly1

```

The loop counter ‘`dc2`’ is being used to control how many times the inner loop is executed.

Note that there is no need to clear the inner loop counter (`dc1`) on each iteration of the outer loop, because every time the inner loop completes, `dc1 = 0`.

The total time taken for each iteration of the outer loop is 1023 cycles for the inner loop, plus 1 cycle for the ‘`decfsz dc2, f`’ and 2 cycles for the ‘`goto`’ at the end, except for the final iteration, which, as we’ve seen, takes 1 cycle less. The three setup instructions at the start add 3 cycles, so the total delay (assuming $N > 0$) is:

$$\text{delay time} = (N \times (1023 + 3) - 1 + 3) \text{ cycles} = (N \times 1026 + 2) \mu\text{s}.$$

The maximum delay would be with 256 outer loop iterations, giving 262,658 μ s. We need a bit less than double that. We could duplicate all the delay code, but it takes fewer lines of code if we only duplicate the inner loop, as shown:

```

        ; delay 500 ms
        movlw    .244                ; outer loop: 244 x (1023 + 1023 + 3) + 2
        movwf   dc2                  ; = 499,958 cycles
        clrf    dc1                  ; inner loop: 256 x 4 - 1
dly1    nop                          ; inner loop 1 = 1023 cycles
        decfsz  dc1,f
        goto   dly1
dly2    nop                          ; inner loop 2 = 1023 cycles
        decfsz  dc1,f
        goto   dly2
        decfsz  dc2,f
        goto   dly1

```

The two inner loops of 1023 cycles each, plus the 3 cycles for the outer loop control instructions (`decfsz` and `goto`) make a total of 2049 μ s. Dividing this into the required 500,000 gives 244.02. This is very close to a whole number, so an outer loop count of 244 will give a good result.

The total execution time for this delay code is 499.958 ms – within 0.01% of the desired result!

Since the internal RC oscillator has a precision of only around $\pm 2\%$, there is no point trying to make this delay any more accurate. But in some cases, to generate a given delay, you will need to add or remove ‘`nop`’ or ‘`goto $+1`’ instructions while adjusting the number of loop iterations. With a little experimentation, it is generally possible to get quite close to the delay you need.

For delays longer than about 0.5 s, you’ll need to add more levels of nesting – with enough levels you generate delays which last for years!

Next we need to be able to toggle, or flip the GP1 output from low to high and back again.

As we saw in [baseline lesson 2](#), to flip a single bit, you can exclusive-or it with 1.

For example, to toggle GP1, we could write:

```

        movlw    1<<GP1              ; bit mask to flip only GP1
        xorwf   GPIO,f               ; flip bits in GPIO

```

The ‘`xorwf`’ instruction exclusive-ors the W register with the specified register – “**exclusive-or W with file register**”, and writes the result either to the specified file register (GPIO in this case) or to W, depending on whether ‘`f`’ or ‘`w`’ is given as the instruction destination.

However, as mentioned earlier, there is a danger in using instructions, such as ‘`xorwf`’, which read from a register, modify the contents and then write the new value back to the register, to operate directly on port registers, because the value read from a port pin will not always be the same as that written to it.

To avoid these potential read-modify-write problems, it is better to use a *shadow register*, which holds a copy of the value the port register is supposed to have, operating on that shadow copy and then copying the updated value to the port register in a single operation.

For example, if we define a variable to use as a shadow register:

```

        UDATA_SHR
sGPIO   res 1                        ; shadow copy of GPIO

```

we can use it in a loop to flash the LED, as follows:

```

        clrf    sGPIO                ; start with shadow GPIO zeroed

```

```

flash    ; toggle LED
         movf    sGPIO,w          ; get shadow copy of GPIO
         xorlw   1<<GP1          ; flip bit corresponding to GP1
         movwf   sGPIO           ; in shadow register
         banksel GPIO           ; and write to GPIO
         movwf   GPIO

         ; delay 500 ms (delay code goes here)

         goto   flash           ; repeat forever

```

The ‘`movf`’ instruction – “**move file register to destination**” – is used to read a register.

With ‘`,w`’ as the destination, ‘`movf`’ copies the contents of the specified register to *W*.

With ‘`,f`’ as the destination, ‘`movf`’ copies the contents of the specified register to itself. That would seem to be pointless; why copy a register back to itself? The answer is that the ‘`movf`’ instruction affects the *Z* (zero) status flag, so copying a register to itself is a way to test whether the value in the register is zero.

The ‘`xorlw`’ instruction exclusive-ors the given literal (constant) value with the *W* register, placing the result in *W* – “**exclusive-or literal to *W***”.

Complete program

Putting together all the above pieces, here’s the complete program for flashing an LED:

```

;*****
;
; Description:      Lesson 1, example 2
;
; Flashes an LED at approx 1 Hz.
; LED continues to flash until power is removed.
;
; Uses inline 500 ms delay routine
;
;*****
;
; Pin assignments:
; GP1 = indicator LED
;
;*****

list      p=12F629
#include   <p12F629.inc>

errorlevel -302          ; no warnings about registers not in bank 0

;***** CONFIGURATION
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4 Mhz int clock
__CONFIG      _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

;***** VARIABLE DEFINITIONS
                UDATA_SHR
sGPIO        res 1          ; shadow copy of GPIO
dc1          res 1          ; delay loop counters
dc2          res 1

```

```

;***** RESET VECTOR *****
RESET    CODE    0x0000        ; processor reset vector
        ; calibrate internal RC oscillator
        call    0x03FF        ; retrieve factory calibration value
        banksel OSCCAL        ; (stored at 0x3FF as a retlw k)
        movwf   OSCCAL        ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start
        ; configure port
        movlw   ~(1<<GP1)     ; configure GP1 (only) as an output
        banksel TRISIO
        movwf   TRISIO

        clrfsz  sGPIO         ; start with shadow GPIO zeroed

;***** Main loop
main_loop
        ; toggle LED
        movf    sGPIO,w       ; get shadow copy of GPIO
        xorlw   1<<GP1        ; toggle bit corresponding to GP1
        movwf   sGPIO         ; in shadow register
        banksel GPIO         ; and write to GPIO
        movwf   GPIO

        ; delay 500 ms
        movlw   .244          ; outer loop: 244 x (1023 + 1023 + 3) + 2
        movwf   dc2           ; = 499,958 cycles
        clrfsz  dc1           ; inner loop: 256 x 4 - 1
dly1    nop                ; inner loop 1 = 1023 cycles
        decfsz  dc1,f
        goto    dly1
dly2    nop                ; inner loop 2 = 1023 cycles
        decfsz  dc1,f
        goto    dly2
        decfsz  dc2,f
        goto    dly1

        ; repeat forever
        goto    main_loop

END

```

If you follow the programming procedure described earlier, you should now have an LED flashing at something very close to 1 Hz.

Conclusion

There has been a lot of theory in this lesson, but we now have a solid base to build on.

By flashing an LED, you have shown that you have a working development environment and that you can create projects, modify your code, load (program) your code into your PIC, and make it run.

We've seen how to toggle a pin, and how to use shadow registers can be used to avoid potentially problematic "read-modify-write" operations on a port.

We also saw how to use decrement instructions with conditional tests to implement loops, and how to use loops to create delays of any length.

In the [next lesson](#) we'll see how to make the code more modular, so that useful code such as the 500 ms delay developed here can be easily re-used within a program, or in other programs.

Introduction to PIC Programming

Mid-Range Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 2: Introducing Modular Code

In [lesson 1](#), we developed a delay routine, which was used in flashing an LED.

This lesson revisits the material introduced in [baseline lesson 3](#), which explored ways in which useful pieces of code, such as delay routines, could be effectively re-used within a program, or in other programs.

This *modular* approach to programming is usually more efficient, because only one copy of a routine is held in memory, and is less likely to introduce errors, because code changes (for example, in the way a delay is implemented) have to be changed in only one place. And, having solved a problem once, you can more easily draw upon your library of existing routines, to include that code into a new program.

You'll save yourself a lot of time if you learn to write re-usable, modular code, which is why it's being covered in such an early lesson.

In summary, this lesson covers:

- Subroutines
- Relocatable code
- External modules
- Banking and paging

Subroutines

The 500 ms delay routine developed in [lesson 1](#) was placed *inline*, within the main loop. If you wished to re-use it in another part of the program, you would need to repeat the whole routine, wasting program memory and making the source code longer than it needs to be. You would have to be careful, when copying and pasting code, to change all of the references to address labels, to avoid your code inadvertently jumping back from the copy to the original routine. And if you wished to change the way the routine was implemented, you would have to find and update every instance of it in the program.

The usual way to use the same routine in a number of places in a program is to place it into a *subroutine*. If we implemented the 500 ms delay as a subroutine, the main loop of the “flash an LED” program would look something like:

```
flash    movf      sGPIO,w          ; get shadow copy of GPIO
         xorlw    1<<GP1          ; flip bit corresponding to GP1
         banksel  GPIO             ; write to GPIO
         movwf   GPIO
         movwf   sGPIO            ; and update shadow copy

         call    delay500         ; delay 500ms

         goto   flash            ; repeat forever
```

The ‘call’ instruction is similar to ‘goto’, in that it jumps to another program address. But first, it copies (or *pushes*) the address of the next instruction onto the stack. The *stack* is a set of registers, used to hold the return addresses of subroutines. When a subroutine is finished, the *return address* is copied (*popped*) from the stack to the program counter, and program execution continues with the instruction following the subroutine call.

Note that the mid-range architecture does **not** suffer from the ‘call’ address limitation, discussed in [baseline lesson 3](#). Subroutine entry points can be placed anywhere in program memory on mid-range PICs; there is no reason to use jump tables, as we did for the baseline devices.

The mid-range PICs have eight stack registers, compared with only two in the baseline architecture. This gives us much more freedom to call subroutines from within other subroutines, making it easier to write larger, more complex programs in a modular way.

In the baseline architecture, the only instruction way to *return* from a subroutine is to use the ‘retlw’ instruction, which returns with a literal in W.

Mid-range PICs, on the other hand, provide a ‘return’ instruction – “**return** from subroutine”, which, as the name suggests, simply returns from a subroutine, without affecting W.

Here then is the 500 ms delay routine, implemented as a subroutine:

```

delay500                                ; delay 500 ms
    movlw    .244                        ; outer loop: 244x(1023+1023+3)-1+3+4
    movwf    dc2                          ;   = 499,962 cycles
    clrf     dc1
dly1    nop                               ; inner loop 1 = 256x4-1 = 1023 cycles
    decfsz   dc1, f
    goto     dly1
dly2    nop                               ; inner loop 2 = 1023 cycles
    decfsz   dc1, f
    goto     dly2
    decfsz   dc2, f
    goto     dly1

    return

```

Example 1: Flash an LED (using delay subroutine with parameter passing)

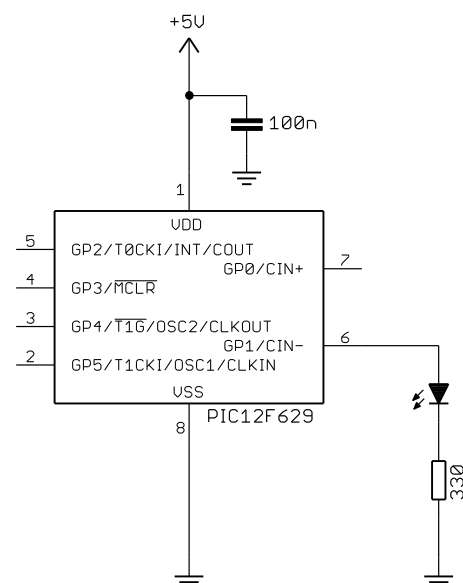
In [lesson 1](#), we used the fixed 500 ms delay routine to flash the LED in this simple circuit (below, with the reset switch and pull-up omitted for clarity) at 1 Hz, with a 50% duty cycle.

But what if we wanted to flash the LED at 1 Hz, with a 20% duty cycle? That is, the LED would be repeatedly turned on for 200 ms, and then off for 800 ms.

You may think that would mean writing two delay routines: one 200 ms and one 800 ms.

But a better, more flexible, approach is to write a single routine, capable of generating a range of delays. The requested delay would be passed as a *parameter* to the delay subroutine.

If you had a number of parameters to pass (for example, a ‘multiply’ subroutine would have to be given the two numbers to multiply), you’d need to place the parameters in general purpose registers, accessed by both the calling program and the subroutine. But if there is only one parameter to pass, it’s often convenient to simply place it in W.



Ideally, we would pass the required delay, in milliseconds, to the routine. But since the mid-range PICs are 8-bit devices, the largest value that can be passed in the *W* register is 255, which is not enough to specify an 800 ms delay.

If the delay routine produces a delay which is some multiple of 10 ms, it could be used for any delay from 10 ms to 2.55 s, which is quite useful – you’ll find that you commonly want delays in this range.

To implement a $W \times 10$ ms delay, we need an inner routine which creates a 10 ms (or close enough) delay, and an outer loop which counts the specified number of those 10 ms loops.

We can count multiples of 10 ms, using a third loop counter, as in the following subroutine:

```
delay10
    movwf    dc3                ; delay = 1+Wx(3+10009+3)-1+4 = W x 10.015ms
dly2    movlw    .13            ; repeat inner loop 13 times
        movwf    dc2            ; -> 13x(767+3)-1 = 10009 cycles
        clrf     dc1            ; inner loop = 256x3-1 = 767 cycles
dly1    decfsz   dc1,f          ;
        goto    dly1
        decfsz   dc2,f          ; end middle loop
        goto    dly1
        decfsz   dc3,f          ; end outer loop
        goto    dly2

        return
```

This routine can then be called, from the main loop, to generate the 200 ms and 800 ms delays we need, as follows:

```
main_loop
    ; turn on LED
    banksel  GPIO
    movlw   1<<GP1        ; set GP1
    movwf   GPIO
    ; delay 0.2 s
    movlw   .20           ; delay 20 x 10 ms = 200 ms
    call    delay10
    ; turn off LED
    clrf    GPIO          ; (clearing GPIO clears GP1)
    ; delay 0.8 s
    movlw   .80           ; delay 80 x 10ms = 800ms
    call    delay10

    ; repeat forever
    goto   main_loop
```

Note that this code does not use a shadow register. It’s no longer necessary, because the **GP1** bit is being set by writing a whole byte to **GPIO**, and is cleared by clearing the whole of **GPIO** in a single operation. It’s not being flipped; there’s no dependency on its previous value. At no time does the **GPIO** register have to be read. It’s only ever being written to. So the “read-modify-write” problem does not apply.

It’s important to understand this point, but if you’re ever in doubt about whether the “read-modify-write” problem may apply, it’s best to play safe and use a shadow register.

We can get away with this approach in this example because **GP1** is the only I/O pin being used. If any of the other pins were being used as outputs, we would have to preserve their value by using instructions which read and modify **GPIO**, in which case a shadow register should be used instead.

You could, if you wish, include a 'banksel GPIO' directive before each instruction which writes to GPIO, but since GPIO is the only banked register accessed within the flash loop, it is ok to select the correct bank at the beginning of the loop.

Complete program

Here is the complete program for flashing the LED with a 20% duty cycle:

```

;*****
;
; Description: Lesson 2, example 1
;
; Flashes an LED at approx 1 Hz, with 20% duty cycle
; LED continues to flash until power is removed.
;
; Uses W x 10 ms delay subroutine
;
;*****
; Pin assignments:
; GP1 = indicator LED
;
;*****

list p=12F629
#include <p12F629.inc>

errorlevel -302 ; no warnings about registers not in bank 0

;***** CONFIGURATION
; ext reset, no code or data protect, no brownout detect,
; no watchdog, power-up timer, 4 Mhz int clock
__CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

;***** VARIABLE DEFINITIONS
UDATA_SHR
dc1 res 1 ; delay loop counters
dc2 res 1
dc3 res 1

;***** RESET VECTOR *****
RESET CODE 0x0000 ; processor reset vector
; calibrate internal RC oscillator
call 0x03FF ; retrieve factory calibration value
banksel OSCCAL ; (stored at 0x3FF as a retlw k)
movwf OSCCAL ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start
; configure port
movlw ~(1<<GP1) ; configure GP1 (only) as an output
banksel TRISIO
movwf TRISIO

```



```

;***** Main loop
main_loop
    ; turn on LED
    banksel GPIO
    movlw    1<<GP1          ; set GP1
    movwf   GPIO
    ; delay 0.2 s
    movlw   .20              ; delay 20 x 10 ms = 200 ms
    call    delay10
    ; turn off LED
    clrf   GPIO              ; (clearing GPIO clears GP1)
    ; delay 0.8 s
    movlw   .80              ; delay 80 x 10ms = 800ms
    call    delay10

    ; repeat forever
    goto   main_loop

;***** SUBROUTINES *****
;***** Variable delay: 10 ms to 2.55 s
;
; Delay = W x 10 ms
;
delay10
    movwf   dc3              ; delay = 1+Wx(3+10009+3)-1+4 = W x 10.015ms
dly2     movlw   .13          ; repeat inner loop 13 times
        movwf   dc2          ; -> 13x(767+3)-1 = 10009 cycles
        clrf   dc1          ; inner loop = 256x3-1 = 767 cycles
dly1     decfsz  dc1,f
        goto   dly1
        decfsz  dc2,f        ; end middle loop
        goto   dly1
        decfsz  dc3,f        ; end outer loop
        goto   dly2

    return

END

```

Relocatable Modules

If you wanted to re-use a subroutine in another program, you could simply copy the subroutine source code into the new program.

There are, however, a few potential problems with this approach:

- Address labels, such as 'dly1', may already be in use in the new program.
- You need to know which variables and macros are needed by the subroutine, and remember to copy their definitions to the new program.
- Variable names, constants and macro definitions have the same problem as address labels – they may already be used in new program, in which case you'd need to identify and rename all references to them.
- The subroutine may need a particular include file; this will need to be identified and included in the new program.

These *namespace* clashes and other problems can be avoided by keeping the subroutine code, along with the variable, constants, macros etc. that it relies on, in a separate source file, where it is assembled into an object file, called an *object module*.

The object modules – one for the main code, plus one for each module, are then linked together to create the final executable code, which is output as a .hex file to be programmed into the PIC.

Banking and Paging

As explained in [baseline lesson 3](#), program memory on baseline PICs is divided into multiple *pages*, each 512 words long, because the `goto` instruction can only specify a 9-bit address. In the baseline architecture, page selection bits in the `STATUS` register make it possible to jump to a program memory location outside the current page.

In mid-range PICs, the longer 14-bit *opcodes* allow both the `goto` and `call` instructions to specify an 11-bit address, or a range of 2048 locations. But since the mid-range PICs can have up to 8192 words of program memory, a paging scheme is still needed to make all of this memory accessible. Thus, in the mid-range architecture, the *page size* is 2048 (or 2k) words, and there can be up to four pages.

The paging scheme is different to that in the baseline architecture.

In the mid-range architecture, the current page is selected by bits 3 and 4 of the `PCLATH` register, which are copied to bits 11 and 12 of the *program counter* (`PC`) whenever a `goto` or `call` instruction is executed.

The lower bits of `PCLATH` are used when a computed `goto` operation is performed, as we will see when table reads are introduced in [lesson 12](#).

[Baseline lesson 3](#) showed how the `'pagesel'` directive can, and should, be used to correctly set the page selection bits, when jumping to an address which may be in a different page. This is just as true for the mid-range architecture as it is for baseline PICs; although the paging mechanism is different, `'pagesel'` is used in the same way.

Page selection is relevant to a discussion of modular code, because the linker may load an object module anywhere in memory; that is why these modules, and this programming style, are described as being *relocatable*. This means that, when calling a subroutine in another module, you will not know if the subroutine is in the current page.

This is also true if you use multiple `CODE` sections within a single source file; unless you place the code sections at a specific address (which is not recommended, since it makes it more difficult for the linker to fit the sections into memory pages), you cannot know where each section will be placed in memory.

Therefore, you should use `pagesel` whenever jumping to or calling a routine in a different code section or module. And note that, after returning from a `call` to a module, the page selection bits will still be set for whatever page that module is in, not necessarily the current page. So it is a good idea to place a `'pagesel $'` directive (“select page for current address”) after each `call` to a subroutine in another module, to ensure that the current page is selected after returning from the subroutine.

You do not, however, need to use `pagesel` before every `goto` or `call`, or after every `call`. Remember that, provided you use the default linker scripts, a single code section is guaranteed to be wholly contained within a single page. Once you know that you've selected the correct page, subsequent `gotos` or `calls` to the same section will work correctly. But be careful!

If in doubt, using `pagesel` before every `goto` and `call` is a safe approach that will always work.

When assembling code for a device, such as the PIC12F629, which has only a single page of program memory, the `pagesel` directive will not generate any object code, so there is no penalty for using it on

PICs where page selection is not an issue. The assembler will, however, warn you that `pagesel` isn't needed on these devices. If you find these messages annoying, you can turn them off with:

```
errorlevel -312 ; no "page or bank selection not needed" messages
```

If you use `pagesel`, even on devices with only a single page of program memory, your code will be more portable, so it is best to always use it, regardless of which mid-range or baseline PIC you are using.

Similarly, when variables are defined in a relocatable module, or if you declare multiple `UDATA` sections within a single source file, you will not know which bank of data memory they are located in.

Therefore, when accessing variables defined in another module or data section, you should use the `'banksel'` directive to correctly set the bank selection bits. Although the data memory bank selection mechanism in the mid-range architecture (described in [lesson 1](#)) differs from that in the baseline PICs (described in [baseline lesson 3](#)), `'banksel'` is used in the same way.

Note that, in the mid-range architecture, where most of the special function registers are banked, the bank selection bits may have to be changed to allow access to a banked SFR. This means that, after having accessed a banked SFR, you may need to select a different bank to access a banked variable. This is different from the baseline architecture where, having selected the bank containing the variables your routine is using, you can access SFRs without having to worry about bank selections. On mid-range PICs, you will not know if a particular set of variables (defined in a `UDATA` section) are in the same bank as any banked SFRs your routine is accessing, so you must use `banksel` when switching between banked SFR and register access.

To summarise:

- The first time you access a variable declared in a `UDATA` section, use `banksel`.
- To access subsequent variables in the same `UDATA` section, you don't need to use `banksel`. (unless you had selected another bank between variable accesses)
- Following a call to a subroutine or external module, which may have selected a different bank, use `banksel` for the first variable accessed after the call.
- Whenever you access a banked special function register, use `banksel`.
- After accessing a banked special function register, use `banksel` when you subsequently access a variable declared in a `UDATA` section
- There is never any need to use `banksel` to access variables in a `UDATA_SHR` section.
- When accessing non-banked special function registers, such as `STATUS`, there is no need to use `banksel`.

Creating a Relocatable Module

Converting an existing subroutine, such as the `'delay10'` routine, into a standalone, relocatable module is easy. All you need to do is to declare any symbols (address labels or variables) that need to be accessible from other modules, using the `GLOBAL` directive.

For example:

```
#include <p12F629.inc> ; any midrange device will do

GLOBAL delay10

;***** VARIABLE DEFINITIONS
        UDATA_SHR
dc1     res 1 ; delay loop counters
```

```

dc2      res 1
dc3      res 1

;***** SUBROUTINES *****
        CODE

;***** Variable delay: 10 ms to 2.55 s
;
; Delay = W x 10 ms
;
delay10
        movwf   dc3                ; delay = 1+Wx(3+10009+3)-1+4 = W x 10.015 ms

dly2    movlw   .13                ; repeat inner loop 13 times
        movwf   dc2                ; -> 13x(767+3)-1 = 10009 cycles
        clrf    dc1                ; inner loop = 256x3-1 = 767 cycles

dly1    decfsz  dc1,f              ;
        goto    dly1
        decfsz  dc2,f              ; end middle loop
        goto    dly1
        decfsz  dc3,f              ; end outer loop
        goto    dly2

        return

        END

```

This is the subroutine from example 1, with a `CODE` directive at the beginning of it, and a `UDATA_SHR` directive to reserve data memory for the subroutine's variables. For most mid-range PICs, with banked memory available, it would be more appropriate to use `UDATA`, to conserve the more valuable shared registers, but since this module is intended to be used with a 12F629, we have to use shared memory. It would make sense to have two versions of this module in your library: one where banked memory is available and one (this version) where it is not.

Toward the start, a `GLOBAL` directive has been added, declaring that the 'delay10' label is to be made available (*exported*) to other modules, so that they can call this subroutine.

You should also add a '`#include`' directive, to define any "standard" symbols used in the code, such as the instruction destinations 'w' and 'f'. This delay routine will work on any mid-range PIC; it's not specific to any, so you can use the include file for any of the mid-range PICs, such as the 12F629. Note that there is no `list` directive; this avoids the processor mismatch errors that would be reported if you specify more than one processor in the modules comprising a single project. You will, however, still see warnings about "Processor-header file mismatch" if your device is different to the processor that the include file is intended for; you can generally ignore these warnings, but, if in doubt, change the '`#include`' directive in the module to match the processor you are building the code for.

Of course it's also important to add a block of comments at the start; they should describe what this module is for, how it is used, any effects (including side effects) it has, and any assumptions that have been made. In this case, it is assumed that the processor is clocked at exactly 4 MHz.

Calling Relocatable Modules

Having created an *external* relocatable module (i.e. one in a separate file), we need to declare, in the main (or *calling*) file any labels we want to use from the external module, so that the linker knows that these labels are defined in another module. That's done with the `EXTERN` directive.

For example:

```

        EXTERN      delay10        ; W x 10ms delay

```

After having been declared as external, it is then possible to call a subroutine or access a variable in an external module (using `pagesel` or `banksel` first!) in the usual way.

To summarise:

- The `GLOBAL` and `EXTERN` directives work as a pair.
- `GLOBAL` is used in the file that defines a module, to export a symbol for use by other modules.
- `EXTERN` is used when calling external modules. It declares that a symbol has been defined elsewhere.

Example 2: Flashing an LED (using an external module)

As we did in [lesson 1](#), we can use the circuit from example 1 to flash an LED at 1 Hz, with a 50% duty cycle – but this time using an external delay module.

The source code for the modular version of the ‘delay10’ routine was given above. You will need to save this as a separate file, called something like ‘delay10.asm’.

A few methods for creating a multiple-file project were described in detail in [baseline lesson 3](#), but, briefly, you need to add the file containing your ‘delay10’ module to your project, which can be done by right-clicking in “Source Files” in the project tree, and then selecting “Add Files” (in MPLAB 8) or “Add Existing Item...” (in MPLAB X) from the context menu.

The main program can be created by copying the “Flash an LED” example from [lesson 1](#), changing the code to call the ‘delay10’ routine (after declaring it to be external), and adding the ‘delay10.asm’ file to your project. You should also update the comments, to state that this external module is required.

Complete program

Here is the main program for flashing an LED with the modifications described above, using the external ‘delay10’ module:

```

;*****
;
;   Filename:      MA_L2-Flash_LED-50p-mod.asm
;   Date:         2/5/12
;   File Version:  1.2
;
;   Author:       David Meiklejohn
;   Company:     Gooligum Electronics
;
;*****
;
;   Architecture:  Midrange PIC
;   Processor:    12F629
;
;*****
;
;   Files required: delay10-shr.asm      (provides W x 10ms delay,
;                                       shared memory version)
;
;*****
;
;   Description:   Lesson 2, example 2
;
;   Demonstrates how to call external modules
;
;   Flashes an LED at approx 1 Hz.
;   LED continues to flash until power is removed.

```

```

;
; Uses W x 10 ms delay module
;
;*****
;
; Pin assignments:
;   GP1 = indicator LED
;*****

list      p=12F629
#include   <p12F629.inc>

errorlevel -302    ; no "register not in bank 0" warnings
errorlevel -312    ; no "page or bank selection not needed" messages

EXTERN    delay10      ; W x 10ms delay

;***** CONFIGURATION
; ext reset, no code or data protect, no brownout detect,
; no watchdog, power-up timer, 4 Mhz int clock
__CONFIG   _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

;***** VARIABLE DEFINITIONS
UDATA_SHR
sGPIO      res      1          ; shadow copy of GPIO

;***** RESET VECTOR *****
RESET      CODE      0x0000    ; processor reset vector
; calibrate internal RC oscillator
call       0x03FF            ; retrieve factory calibration value
banksel   OSCCAL            ; (stored at 0x3FF as a retlw k)
movwf     OSCCAL            ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start
; configure port
movlw     ~(1<<GP1)          ; configure GP1 (only) as an output
banksel   TRISIO
movwf     TRISIO

clrf      sGPIO              ; start with shadow GPIO zeroed

;***** Main loop
main_loop
; toggle LED
movf      sGPIO,w            ; get shadow copy of GPIO
xorlw     1<<GP1              ; toggle bit corresponding to GP1
movwf     sGPIO              ; in shadow register
banksel   GPIO               ; and write to GPIO
movwf     GPIO

```

```
; delay 500 ms -> 1 Hz flashing at 50% duty cycle
movlw    .50
pagesel  delay10          ; delay 50 x 10 ms = 500 ms
call     delay10

; repeat forever
pagesel  main_loop
goto    main_loop

END
```

Conclusion

Again, this has been a lot theory – and we’re still only flashing an LED!

The intent of this lesson was to give you an understanding of the mid-range PIC memory architecture, including its limitations (banking and paging) and how to work around them, to avoid potential problems as your programs grow. We’ve also seen how to create re-usable code modules, which should help you to avoid wasting time “reinventing the wheel” for each new project in future. In fact, we’ll continue to use the delay module in later lessons.

In the [next lesson](#) we’ll look at reading and responding to switches, such as pushbuttons. And since real switches “bounce”, which can be a problem for microcontroller applications, we’ll look at ways to “debounce” them.

Introduction to PIC Programming

Mid-Range Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 3: Reading Switches

The [first lesson](#) introduced simple digital output, by turning on or flashing an LED. That's more useful than you may think, since, with some circuit changes (such as adding transistors and relays), it can be readily adapted to turning on and off almost any electrical device.

Most systems, however, need to interact with their environment in some way; to respond to user commands or varying inputs. The simplest form of input is an on/off switch. This lesson revisits the material covered in [baseline lesson 4](#), showing how to read a simple pushbutton switch – techniques which are applicable to any digital (strictly on/off or high/low) input.

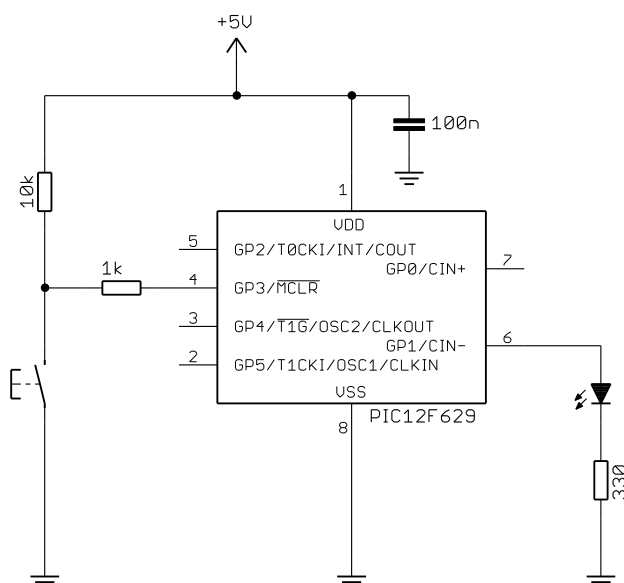
This lesson covers:

- Reading digital inputs
- Conditional branching
- Using internal pull-ups
- Software approaches to switch debouncing

Example 1: Reading a Digital Input

One of the simplest ways to generate a digital input signal is to use a basic pushbutton switch.

The [Gooligum training board](#) provides pushbutton switches connected to pins GP2 and GP3, while the Microchip Low Pin Count demo board only has a pushbutton connected to GP3, as in the circuit (from [lesson 1](#)) shown below, so we'll use it again in this lesson.



As before, we'll use the LED on GP1 as an indicator.

If you're using the Gooligum training board, you should close jumpers JP3 and JP12, to bring the 10 kΩ resistor into the circuit and to connect the LED to GP1.

The 10 kΩ resistor is a *pull-up* resistor, holding GP3 high while the switch is open.

When the switch is pressed, the pin is pulled to ground through the 1 kΩ resistor.

Given the high impedance of the PIC's inputs (very little current flows into them), these external resistors are sufficient to pull the input voltage to a valid logic high when the pushbutton is up, and a valid logic low when it is pressed. For a more detailed analysis, see [baseline lesson 4](#).

Interference from $\overline{\text{MCLR}}$

There is a potential problem with using a pushbutton on GP3 because that pin is also used for $\overline{\text{MCLR}}$ and, as we saw in [baseline lesson 1](#), MPLAB provides for control of the $\overline{\text{MCLR}}$ line through the “Release from Reset” and “Hold in Reset” menu items (MPLAB 8 only) and toolbar buttons (MPLAB 8 and MPLAB X).

That's not a problem if you're using a PICkit 3, because when the PICkit 3 is used as a programmer¹, its $\overline{\text{MCLR}}$ output is disconnected (“tri-stated”) immediately after programming, meaning that the PICkit 3 won't affect the PIC's $\overline{\text{MCLR}}$ / GP3 input.

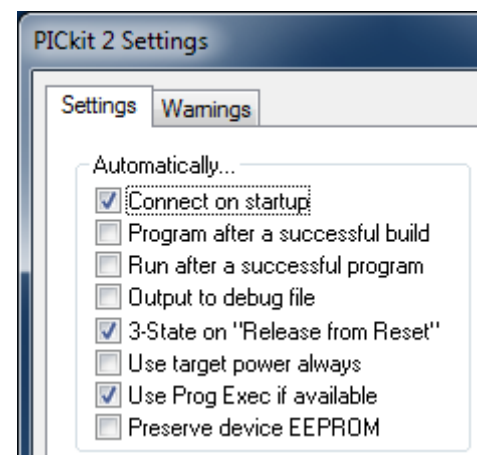
It's different with the PICkit 2 where, by default, the PICkit 2 continues to assert control over the $\overline{\text{MCLR}}$ line and, because of the 1 k Ω isolation resistor, the 10 k Ω pull-up resistor and the pushbutton cannot overcome the PICkit 2's control of that line.


When the PICkit 2 is used as a programmer with MPLAB, it will, by default, assert control of the $\overline{\text{MCLR}}$ line, overriding any pushbutton switch on the PIC's $\overline{\text{MCLR}}$ / GP3 input.

If you are using MPLAB 8, this problem can be overcome by changing the PICkit 2 programming settings, to tri-state the PICkit 2's $\overline{\text{MCLR}}$ output (effectively disconnecting it) when it is not being used to hold the PIC in reset.

To do this, select the PICkit 2 as a programmer (using the “Programmer → Select Programmer” submenu) and then use the “Programmer → Settings” menu item to display the PICkit 2 Settings dialog window, shown on the right. Select “3-State on “Release from Reset”” in the Settings tab and then click “OK”.

After using the PICkit 2 to program your device, it will hold $\overline{\text{MCLR}}$ low, holding the GP3 input low, overriding the pull-up resistor.



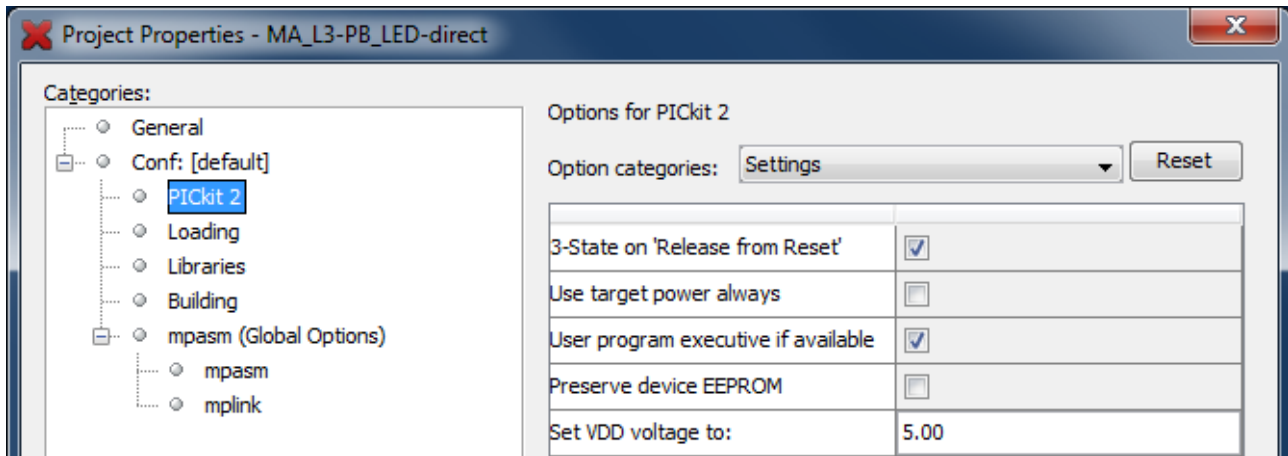
When you now click on the  button in the programming toolbar, or select the “Programmer → Release from Reset” menu item, the PICkit 2 will release control of $\overline{\text{MCLR}}$, allowing GP3 to be driven high or low by the pull-up resistor and pushbutton.

MPLAB X also allows you to prevent the PICkit 2 asserting control over $\overline{\text{MCLR}}$, in much the same way.

¹ as opposed to being used as a debugger; see [lesson 0](#)

To do this, open the Project Properties window, by selecting the “File → Project Properties” menu item, or right-clicking the project name in the Projects window and selecting “Properties”, or simply clicking on the “Project Properties” button to the left of the Dashboard.

If you click on “PICkit 2”, you will see the settings shown below:



Select “3-State on ‘Release from Reset’”, and then click “OK”.

Now, when you build and run your project, the PICkit 2’s $\overline{\text{MCLR}}$ output will be tri-stated, making it possible for you to use a pushbutton on GP3.

Reading the Switch

We’ll start with a short program that simply turns the LED on when the pushbutton is pressed.

Of course, that’s a waste of a microcontroller. To get the same effect, you could leave the PIC out and build the circuit shown on the right! But, this simple example avoids having to deal with the problem of switch contact bounce, which we’ll look at later.



In general, to read a pin, we need to:

- Configure the pin as an input
- Read or test the bit corresponding to the pin

Recall, from [lesson 1](#), that the pins on the 12F629 are *digital* inputs or outputs. They can be turned on or off, but nothing in between. Similarly, they can read only a voltage as being “high” or “low”. The data sheet defines input voltage ranges where the pin is guaranteed to read as “high” or “low”. For voltages between these ranges, the pin might read as either; the input behaviour for intermediate voltages is *undefined*.

As you might expect, a “high” input voltage reads as a ‘1’, and a “low” reads as a ‘0’.

Normally, to configure a pin as an input, you would set the corresponding TRISIO bit to ‘1’.

This circuit uses GP3, which, because it shares a pin with $\overline{\text{MCLR}}$, can only ever be an input – regardless of the contents of TRISIO. However, when using GP3 as an input, you may as well set bit 3 of TRISIO, to make your code clearer.

An instruction such as `movf GPIO,w` will read the bit corresponding to GP3. The problem with that is that it reads all the pins in GPIO, not just GP3. If you want to act only on a single bit, you need to separate it from the rest, which can be done with logical masking and shift instructions, but there’s a much easier way – use the bit test instructions.

There are two:

'btfsc f,b' tests bit 'b' in register 'f'. If it is '0', the following instruction is skipped – “bit test file register, skip if clear”.

'btfss f,b' tests bit 'b' in register 'f'. If it is '1', the following instruction is skipped – “bit test file register, skip if set”.

Their use is illustrated in the following code:

```

; configure port
movlw  ~(1<<GP1)      ; configure GP1 (only) as an output
banksel TRISIO      ; (GP3 is an input)
movwf  TRISIO
banksel GPIO
clrf   GPIO          ; start with GPIO clear (GP1 low)

;***** Main loop
main_loop
; turn on LED only if button pressed
btfss  GPIO,GP3      ; if button pressed (GP3 low)
bsf    GPIO,GP1      ; turn on LED
btfsc  GPIO,GP3      ; if button up (GP3 high)
bcf    GPIO,GP1      ; turn off LED

; repeat forever
goto   main_loop

```

Note that the logic seems to be inverse; the LED is turned on if GP3 is clear, yet the 'btfss' instruction tests for the GP3 bit being set. Since the bit test instructions skip the next instruction if the bit test condition is met, the instruction following a bit test is executed only if the condition is *not* met. Often, following a bit test instruction, you'll place a 'goto' or 'call' to jump to a block of code that is to be executed if the bit test condition is not met. In this case, there is no need, as the LED can be turned on or off with single bit set or clear instructions.

However, as discussed in [lesson 1](#), directly setting or clearing individual bits in an I/O port can lead to unintended effects, due the potential for read-modify-write problems – you may find that bits other than the designated one are also being changed. This unwanted effect often occurs when sequential bit set/clear instructions are performed on the same port. Trouble can often be avoided by separating sequential 'bsf' and 'bcf' instructions with a 'nop'.

Although unlikely to be necessary in this case, since the bit set/clear instructions are not sequential, a shadow register could be used as follows:

```

; configure port
movlw  ~(1<<GP1)      ; configure GP1 (only) as an output
banksel TRISIO      ; (GP3 is an input)
movwf  TRISIO

banksel GPIO
clrf   GPIO          ; start with GPIO clear (LED off)
clrf   sGPIO         ; update shadow copy

;***** Main loop
main_loop
; turn on LED only if button pressed
btfss  GPIO,GP3      ; if button pressed (GP3 low)
bsf    sGPIO,GP1     ; turn on LED
btfsc  GPIO,GP3      ; if button up (GP3 high)
bcf    sGPIO,GP1     ; turn off LED
movf   sGPIO,w       ; copy shadow to GPIO
movwf  GPIO

```

```

; repeat forever
goto    main_loop

```

It's possible to optimise this a little. There is no need to test for button up as well as button down; it will be either one or the other, so we can instead write a value to the shadow register, assuming the button is up (or down), and then test just once, updating the shadow if the button is found to be down (or up).

The main loop then becomes:

```

main_loop
; turn on LED only if button pressed
clrf    sGPIO          ; assume button up -> LED off
banksel GPIO
btfss   GPIO,GP3       ; if button pressed (GP3 low)
bsf     sGPIO,GP1      ; turn on LED

movf    sGPIO,w        ; copy shadow to GPIO
movwf   GPIO

; repeat forever
goto    main_loop

```

It's also not really necessary to initialise **GPIO** at the start; whatever state it is in when the program starts, it will be updated the first time the loop completes, a few μ s later – much too fast to see. If setting the initial values of output pins correctly is important, to avoid power-on glitches that may affect circuits connected to them, the correct values should be written to the port registers before configuring the pins as outputs, i.e. initialise **GPIO** before **TRISIO**. But when dealing with human perception, it's not important.

If you didn't use a shadow register, but tried to take the same approach – assuming a state (e.g. “button up”), setting **GPIO**, then reading the button and changing **GPIO** accordingly – it would mean that the LED would be flickering on and off, albeit too fast to see. Using a shadow register is a neat solution that avoids this problem, as well as any read-modify-write concerns, since the physical register (**GPIO**) is only ever updated with the correctly determined value.

Complete program

Here is the complete program for turning on the LED when the pushbutton is pressed, using the optimised shadow register code above:

```

;*****
;
; Description:      Lesson 3, example 1b
;
; Demonstrates reading a switch
; (using shadow register to update port)
;
; Turns on LED when pushbutton on GP3 is pressed
;
;*****
;
; Pin assignments:
; GP1 = LED
; GP3 = pushbutton switch (active low)
;
;*****

list          p=12F629
#include      <p12F629.inc>

errorlevel   -302          ; no warnings about registers not in bank 0

```

```

;***** CONFIGURATION
                ; int reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4 Mhz int clock
    __CONFIG    _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
    _PWRTE_ON & _INTRC_OSC_NOCLKOUT

;***** VARIABLE DEFINITIONS
    UDATA_SHR
sGPIO    res    1                ; shadow copy of GPIO

;***** RESET VECTOR *****
RESET    CODE    0x0000            ; processor reset vector
        ; calibrate internal RC oscillator
        call    0x03FF            ; retrieve factory calibration value
        banksel OSCCAL            ; (stored at 0x3FF as a retlw k)
        movwf   OSCCAL            ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start
        ; configure port
        movlw   ~(1<<GP1)        ; configure GP1 (only) as an output
        banksel TRISIO            ; (GP3 is an input)
        movwf   TRISIO

;***** Main loop
main_loop
        ; turn on LED only if button pressed
        clrf    sGPIO            ; assume button up -> LED off
        banksel GPIO
        btfss   GPIO,GP3         ; if button pressed (GP3 low)
        bsf     sGPIO,GP1        ; turn on LED

        movf    sGPIO,w          ; copy shadow to GPIO
        movwf   GPIO

        ; repeat forever
        goto    main_loop

END

```

Example 2: Debouncing

In most applications, you want your code to respond to transitions; some action should be triggered when a button is pressed or a switch is toggled. This presents a problem when interacting with real, physical switches, because their contacts *bounce*. When most switches change, the contacts in the switch will make and break a number of times before settling into the new position. This contact bounce is generally too fast for the human eye to see, but microcontrollers are fast enough to react to each of these rapid, unwanted transitions.

A similar problem can be caused by *electromagnetic interference (EMI)*. Unwanted spikes may appear on an input line, due to electromagnetic noise, especially (but not only) when switches or sensors are some distance

from the microcontroller. But any solution which deals effectively with contact bounce will generally also remove or ignore input spikes caused by EMI.

Dealing with these problems is called switch *debouncing*.

To illustrate the problem, suppose that you wish to toggle the LED on GP1, once, each time the button on GP3 is pressed.

In pseudo-code, this could be expressed as:

```
do forever
    wait for button press
    toggle LED
    wait for button release
end
```

Note that it is necessary to wait for the button to be released before restarting the loop, so that the LED should only toggle once per button press. If we didn't wait for the button to be released before continuing, the LED would continue to toggle as long as the button was held down; not the desired behaviour.

Here is some code which implements this:

```
main_loop
    ; wait for button press
wait_dn btfsc    GPIO,GP3        ; wait until GP3 low
        goto    wait_dn

        ; toggle LED
movf    sGPIO,w
xorlw   1<<GP1                ; toggle bit corresponding to GP1
movwf   sGPIO                  ; in shadow register
movwf   GPIO                   ; and write to GPIO

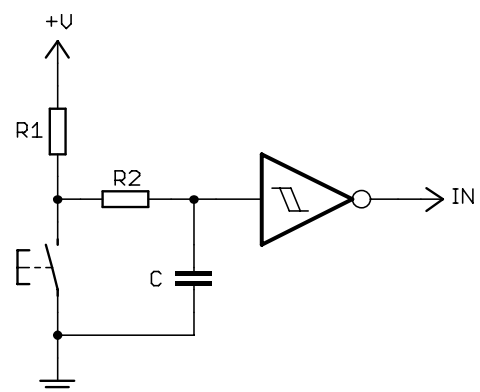
        ; wait for button release
wait_up btfss    GPIO,GP3        ; wait until GP3 high
        goto    wait_up

        ; repeat forever
goto    main_loop
```

If you build this into a complete program² and test it, you will find that it is difficult to reliably change the LED when you press the button; sometimes it will change, other times not. This is due to contact bounce.

In [baseline lesson 4](#) we saw that switch debouncing is in effect a filtering problem and that it can be addressed by using appropriate hardware.

One solution is to use an RC low-pass filter coupled to a Schmitt trigger buffer, as shown on the right.



² You'd need to add processor configuration, reset vector, initialisation code etc., and declare the `sGPIO` variable, as we've done before. The complete source code is provided with these tutorials.

However, one of the reasons to use microcontrollers is that they allow you to solve what would otherwise be a hardware problem, in software. In particular, it is possible to use software routines to debounce a switch input, without any need for external filtering hardware.

If the software can ignore input transitions due to contact bounce or EMI, while detecting and responding to genuine switch changes, no external debounce circuitry is needed. As with the hardware approach, the problem is essentially one of filtering; we need to ignore any transitions too short to be ‘real’.

Debouncing using delays

The easiest approach to software debouncing is to estimate the maximum time the switch could possibly take to settle, and then simply wait at least that long, after detecting the first transition. If the wait time, or delay, is longer than the maximum possible settling time, you can be sure that, by the time the delay completes, the switch will have finished bouncing.

It’s simply a matter of adding a suitable debounce delay, after each transition is detected, as in the following pseudo-code:

```
do forever
    wait for button press
    toggle LED
    delay debounce_time
    wait for button release
    delay debounce_time
end
```

Note that the LED is toggled immediately after the button press is detected. There’s no need to wait for debouncing. By acting on the button press as soon as it is detected, the user will experience as fast a response as possible.

The necessary minimum delay time depends on the characteristics of the switch. For example, the switch tested in baseline lesson 4 was seen to settle in around 250 μ s. Repeated testing showed no settling time greater than 1 ms, but it’s difficult to be sure of that, and perhaps a different switch, say that used in production hardware, rather than the prototype, may behave differently. So it’s best to err on the safe side, and choose the longest delay we can get away with. People don’t notice delays of 20 ms or less (flicker is only barely perceptible at 50 Hz, corresponding to a 20 ms delay), so a good choice is probably 20 ms.

As you can see, choosing a suitable debounce delay is not an exact science!

We can simply add delays, using the $W \times 10$ ms delay module developed in [lesson 2](#), to the main loop of the “Toggle an LED” code (presented above), as follows:

```
main_loop
    ; wait for button press
    banksel GPIO
wait_dn btfsc  GPIO,GP3          ; wait until GP3 low
        goto   wait_dn

        ; toggle LED
        movf   sGPIO,w
        xorlw  1<<GP1           ; toggle bit corresponding to GP1
        movwf  sGPIO           ; in shadow register
        movwf  GPIO           ; and write to GPIO

        ; delay to debounce button press
        movlw  .2
        pagesel delay10
        call   delay10         ; delay 2 x 10 ms = 20 ms
        pagesel $
```

```

        ; wait for button release
        banksel GPIO
wait_up btfss   GPIO,3           ; wait until GP3 high
        goto    wait_up

        ; delay to debounce button press
        movlw   .2
        pagesel delay10
        call    delay10         ; delay 2 x 10 ms = 20 ms
        pagesel $

        ; repeat forever
        goto    main_loop

```

Note the extra ‘banksel’ directives; these have been added in case the ‘delay10’ routine changes the current bank selection. That’s not strictly necessary in this case, because we know that this version of the ‘delay10’ routine does not affect the current bank selection (it only uses shared registers). But in general it’s safer to assume that, when you call a subroutine, it may change both the bank and page selection bits (hence the ‘pagesel \$’ directive following each call to the delay routine – it ensures that the subsequent ‘goto’ instructions in the routine will work correctly).

If you build and test this code, you should find that the LED now reliably changes state every time you press the button.

Debouncing using a counting algorithm

There are a couple of problems with using a fixed length delay for debouncing.

Firstly, the need to be “better safe than sorry” means making the delay as long as possible, and probably slowing the response to switch changes more than is really necessary, potentially affecting the feel of the device you’re designing.

More importantly, the delay approach cannot differentiate between a glitch and the start of a switch change. As discussed, spurious transitions can be caused by EMI, or electrical noise – or a momentary change in pressure while a button is held down.

A commonly used approach, which avoids these problems, is to regularly read (or *sample*) the input, and only accept that the switch is in a new state, when the input has remained in that state for some number of times in a row. If the new state isn’t maintained for enough consecutive times, it’s considered to be a glitch or a bounce, and is ignored.

For example, you could sample the input every 1 ms, and only accept a new state if it is seen 10 times in a row; i.e. high or low for a continuous 10 ms.

To do this, set a counter to zero when the first transition is seen. Then, for each sample period (say every 1 ms), check to see if the input is still in the desired state and, if it is, increment the counter before checking again. If the input has changed state, that means the switch is still bouncing (or there was a glitch), so the counter is set back to zero and the process restarts. The process finishes when the final count is reached, indicating that the switch has settled into the new state.

The algorithm can be expressed in pseudo-code as:

```

count = 0
while count < max_samples
    delay sample_time
    if input = required_state
        count = count + 1
    else
        count = 0
end

```


Here is the modified “toggle an LED” main loop, illustrating the use of this counting debounce algorithm:

```

main_loop
    ; wait for button press
db_dn   clrfsz  db_cnt      ; wait until button pressed (GP3 low)
        clrfsz  dcl        ; debounce by counting:
dn_dly  incfsz  dcl,f      ; delay 256x3 = 768 us.
        goto    dn_dly
        btfsc   GPIO,GP3   ; if button up (GP3 high),
        goto    db_dn      ; restart count
        incf    db_cnt,f    ; else increment count
        movlw   .13        ; max count = 10ms/768us = 13
        xorwf   db_cnt,w    ; repeat until max count reached
        btfss  STATUS,Z    ;
        goto    dn_dly

    ; toggle LED
        movf    sGPIO,w
        xorlw   1<<GP1     ; toggle LED on GP1
        movwf   sGPIO      ; using shadow register
        movwf   GPIO

    ; wait for button release
db_up   clrfsz  db_cnt      ; wait until button released (GP3 high)
        clrfsz  dcl        ; debounce by counting:
up_dly  incfsz  dcl,f      ; delay 256x3 = 768 us.
        goto    up_dly
        btfss  GPIO,GP3   ; if button down (GP3 low),
        goto    db_up      ; restart count
        incf    db_cnt,f    ; else increment count
        movlw   .13        ; max count = 10ms/768us = 13
        xorwf   db_cnt,w    ; repeat until max count reached
        btfss  STATUS,Z    ;
        goto    up_dly

    ; repeat forever
        goto    main_loop

```

There are two debounce routines here; one for the button press, the other for button release. The program first waits for a pushbutton press, debounces the press, then toggles the LED before waiting for the pushbutton to be released, and then debouncing the release.

The only difference between the two debounce routines is the input test: ‘btfsc GPIO, 3’ when testing for button up, versus ‘btfss GPIO, 3’ to test for button down.

Note that, in each of the debounce routines, a short loop is used to generate a 768 µs delay, so the input is being sampled every 768 µs or so, instead of the 1 ms sample time mentioned above – simply because it’s much easier to generate a 768 µs delay than a 1 ms delay. The principle is the same; instead of sampling the input 10 times, 1 ms apart, the routine samples 13 times, 768 µs apart. Either way, the routine is checking that the switch is remaining in the same state (continuously on or off) for approximately 10 ms.

The code above demonstrates one method for counting up to a given value (13 in this case):

The count is zeroed at the start of the routine.

It is incremented within the loop, using the ‘incf’ instruction – “**increment file register**”. As with many other instructions, the incremented result can be written back to the register, by specifying ‘, f’ as the destination, or to W, by specifying ‘, w’ – but normally you would use it as shown, with ‘, f’, so that the count in the register is incremented. The mid-range PICs also provide a ‘decf’ instruction – “**decrement file register**”, which is similar to ‘incf’, except that it performs a decrement instead of increment.

We've seen the 'xorwf' instruction before, but not used in quite this way. The result of exclusive-oring any binary number with itself is zero. If any dissimilar binary numbers are exclusive-ored, the result will be non-zero. Thus, XOR can be used to test for equality, which is how it is being used here. First, the maximum count value is loaded into W, and then this max count value in W is xor'd with the loop count. If the loop counter has reached the maximum value, the result of the XOR will be zero. We do not care what the result of the XOR actually is, only whether it is zero or not. And to avoid overwriting the loop counter with the result, 'w' is specified as the destination of the 'xorwf' instruction – writing the result to W, effectively discarding it.

To check whether the result of the XOR was zero (which will be true if the count has reached the maximum value), we use the 'btfss' instruction to test the zero flag bit, Z, in the STATUS register.

Alternatively, each debounce routine could have been coded by initialising the loop counter to the maximum value at the start of the loop, and using 'decfsz' at the end of the loop, as follows:

```

; wait for button press, debounce by counting:
db_dn  movlw   .13           ; max count = 10ms/768us = 13
        movwf  db_cnt
        clrf   dcl
dn_dly  incfsz  dcl,f         ; delay 256x3 = 768 us.
        goto  dn_dly
        btfsc  GPIO,GP3     ; if button up (GP3 high),
        goto  db_dn         ; restart count
        decfsz db_cnt,f     ; else repeat until max count reached
        goto  dn_dly

```

That's two instructions shorter, and at least as clear, so it's a better way to code this routine.

Nevertheless it's worth knowing how to count up to a given value, using XOR to test for equality, because sometimes it simply makes more sense to count up than down.

Example 3: Internal Pull-ups

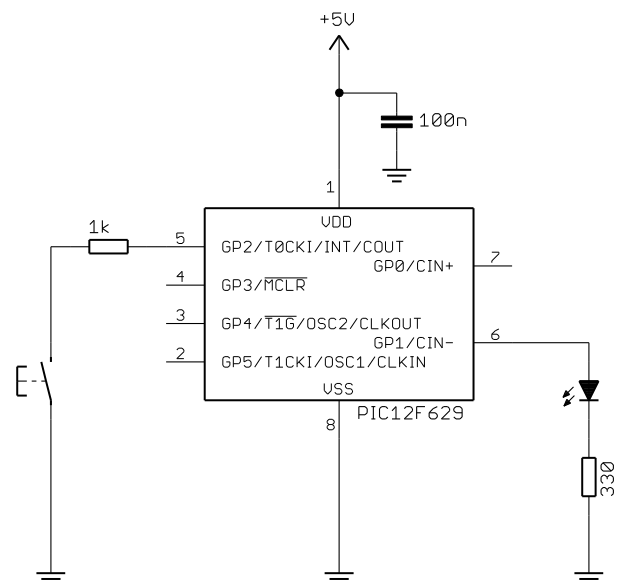
The use of pull-up resistors is so common that most modern PICs make them available internally, on at least some of the pins.

The availability of internal pull-ups makes it possible to do without with the external pull-up resistor, as shown in the circuit on the right.

Unfortunately, there is no internal pull-up on the 12F629's GP3 pin, so to demonstrate their use we need to use a different input pin, which is why the switch is connected to GP2 here.

The [Gooligum training board](#) already has a pushbutton switch connected to GP2 as shown, but you should ensure that jumper JP7 is not closed, so that there is no external pull-up in place.

If you are using the Microchip demo board, you will need to supply your own pushbutton and connect it between GP2 (pin 9 of the 14-pin header) and ground (pin 14 on the header).



Strictly speaking, the internal pull-ups are not simple resistors. Microchip refers to them as “weak pull-ups”; they provide a small current (typically 250 μA) which is enough to hold a disconnected, or *floating*, input high, but not enough to strongly resist any external signal trying to drive the input low.

We’ve seen that, on baseline PICs, internal pull-ups are only available on a few pins, and they are either all enabled or all disabled.

This is different in the mid-range architecture: pull-ups are available for every pin on the 12F629 (except GP3), and they can be selected individually.

Nevertheless, the internal weak pull-ups are globally controlled, as a group, by the $\overline{\text{GPPU}}$ bit in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION_REG	$\overline{\text{GPPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0

By default (after a power-on or reset), $\overline{\text{GPPU}} = 1$ and all the internal pull-ups are disabled.

To globally enable internal pull-ups, clear $\overline{\text{GPPU}}$.

Note that, in the mid-range architecture, the OPTION register is accessed as a normal, memory-mapped register, called OPTION_REG, as mentioned in lesson 1.

*Note: The `option` instruction is **not** used to write to the OPTION register on mid-range devices. The OPTION register is accessed as `OPTION_REG`, using general instructions, such as `bsf`.*

Each weak pull-up is individually controlled by a bit in the WPU register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
WPU	-	-	WPU5	WPU4	-	WPU2	WPU1	WPU0

If $\text{WPU}\langle n \rangle = 1$, the weak pull-up on the corresponding GPIO pin, GPn , is enabled.

If $\text{WPU}\langle n \rangle = 0$, the corresponding weak pull-up is disabled.

However, if a pin is configured as an output, the internal pull-up is automatically disabled for that pin.

To enable the pull-up on GP2, we must first clear $\overline{\text{GPPU}}$ to globally enable weak pull-ups:

```
bcf    OPTION_REG, NOT_GPPU    ; enable global pull-ups
```

One advantage of the mid-range architecture is that, because the OPTION register can be accessed directly, it is possible to clear or set individual bits, such as $\overline{\text{GPPU}}$, leaving the other bits unchanged. On the baseline PICs, we had to load the whole OPTION register in a single operation, which is much less convenient.

Then, having globally enabled weak pull-ups, we need to specifically enable the pull-up on GP2, by setting $\text{WPU}\langle 2 \rangle$.

You could do that by:

```
bsf    WPU, GP2                ; enable pull-up on GP2
```

By default (after a power-on reset), every bit of WPU is set, so there is not really any need to explicitly set WPU<2> like this. But it's good practice to disable the weak pull-ups on the unused input pins (unused inputs should not be left floating, to avoid large current consumption and ESD damage to the PIC, and are often tied to ground; if pull-ups were enabled on grounded inputs, current will flow through them, leading to unnecessary power consumption). Therefore, all the remaining bits in WPU should be cleared.

This could be done by:

```
    clrf    WPU                ; disable all pull-ups
    bsf    WPU,GP2            ; except on GP2
```

or:

```
    movlw  1<<GP2            ; enable pull-up on GP2 only
    movwf  WPU
```

The second form is better if you need to enable pull-ups on more than one input.

Complete program

Here's the complete "Toggle an LED" program, illustrating how to read and debounce a simple switch on a pin held high by an internal pull-up:

```

;*****
;
; Description:      Lesson 3, example 3
;
; Demonstrates use of internal pullups plus debouncing
;
; Toggles LED when pushbutton is pressed then released,
; using a counting algorithm to debounce switch
;
;*****
;
; Pin assignments:
;   GP1 = LED
;   GP2 = pushbutton switch (active low)
;
;*****

list      p=12F629
#include   <p12F629.inc>

errorlevel -302                ; no warnings about registers not in bank 0

;***** CONFIGURATION
                ; int reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4 Mhz int clock
__CONFIG    _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

;***** VARIABLE DEFINITIONS
                UDATA_SHR
sGPIO      res 1                ; shadow copy of GPIO
db_cnt     res 1                ; debounce counter
dc1        res 1                ; delay counter

```

```

;***** RESET VECTOR *****
RESET   CODE    0x0000           ; processor reset vector
        ; calibrate internal RC oscillator
        call    0x03FF           ; retrieve factory calibration value
        banksel OSCCAL           ; (stored at 0x3FF as a retlw k)
        movwf  OSCCAL           ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start
        ; configure port
        movlw  ~(1<<GP1)         ; configure GP1 (only) as an output
        banksel TRISIO           ; (GP2 is an input)
        movwf  TRISIO
        banksel OPTION_REG       ; enable global pull-ups
        bcf    OPTION_REG,NOT_GPPU
        movlw  1<<GP2            ; enable pull-up on GP2 only
        banksel WPU
        movwf  WPU
        banksel GPIO             ; start with LED off
        clrf   GPIO
        clrf   sGPIO            ; update shadow

;***** Main loop
main_loop
        ; wait for button press, debounce by counting:
db_dn   movlw  .13                ; max count = 10ms/768us = 13
        movwf  db_cnt
        clrf   dcl
dn_dly  incfsz  dcl,f              ; delay 256x3 = 768 us.
        goto  dn_dly
        btfsc  GPIO,GP2           ; if button up (GP2 high),
        goto  db_dn               ; restart count
        decfsz db_cnt,f           ; else repeat until max count reached
        goto  dn_dly

        ; toggle LED on GP1
        movf   sGPIO,w
        xorlw  1<<GP1             ; toggle LED on GP1
        movwf  sGPIO             ; using shadow register
        movwf  GPIO

        ; wait for button release, debounce by counting:
db_up   movlw  .13                ; max count = 10ms/768us = 13
        movwf  db_cnt
        clrf   dcl
up_dly  incfsz  dcl,f              ; delay 256x3 = 768 us.
        goto  up_dly
        btfss  GPIO,GP2           ; if button down (GP2 low),
        goto  db_up               ; restart count
        decfsz db_cnt,f           ; else repeat until max count reached
        goto  up_dly

        ; repeat forever
        goto  main_loop

END

```

Conclusion

After working through this lesson, you should be able to write programs which read and respond to simple switches or other digital inputs, and be able to effectively debounce switch or other noisy inputs.

But that's enough on reading switches for now. There's plenty more to explore, of course, such as reading keypads and debouncing multiple switch inputs – topics to explore later.

In the [next lesson](#) we'll look at the PIC12F629's 8-bit timer module, Timer0.

Introduction to PIC Programming

Mid-Range Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 4: Using Timer0

The lessons until now have covered the essentials of mid-range PIC microcontroller operation: controlling digital outputs, timed via programmed delays, with program flow responding to digital inputs. That's enough to allow you to perform a great many tasks. But PICs (and most other microcontrollers) offer a number of additional features that make many tasks much easier. Possibly the most useful of all are *timers*; so useful that at least one is included in every current 8-bit PIC.

A timer is simply a counter, which increments automatically. It can be driven by the processor's instruction clock, in which case it is referred to as a *timer*, incrementing at some predefined, steady rate. Or it can be driven by an external signal, where it acts as a *counter*, counting transitions on an input pin. Either way, the timer continues to count, independently, while the PIC performs other tasks.

And that is why timers are so very useful. Most programs need to perform a number of concurrent tasks; even something as simple as monitoring a switch while flashing an LED. The execution path taken within a program will generally depend on real-world inputs. So it is very difficult in practice to use programmed delay loops, as in [lesson 1](#), to accurately measure elapsed time. But a timer will keep counting, steadily, while your program responds to inputs, performs calculations, or whatever.

As we'll see in [lesson 6](#), timers are commonly used to drive *interrupts* (routines which interrupt the normal program flow) to allow regularly timed "background" tasks to run. However, before moving on to timer-based interrupts, it's important to understand how timers operate. And, as this lesson will demonstrate, timers can be very useful, even when not used with interrupts.

This lesson revisits the material in [baseline lesson 5](#), covering:

- Introduction to the Timer0 module
- Creating delays with Timer0
- Debouncing via Timer0
- Using Timer0 counter mode with an external clock
(demonstrating the use of a crystal oscillator as a time reference)

Timer0 Module

Mid-range PICs can have up to three timers; the simplest of these is referred to as Timer0. The visible part is a single 8-bit register, TMR0, which holds the current value of the timer. It is readable and writeable. If you write a value to it, the timer is reset to that value and then starts incrementing from there.

When it has reached 255, it rolls over to 0, sets an "overflow flag" (the TOIF bit in the INTCON register, triggering an interrupt if Timer0 interrupts are enabled) to indicate that the rollover happened, and then continues to increment.

Note that this is different from the Timer0 module in the baseline architecture, which does not have an overflow flag.

The configuration of Timer0 is set by a number of bits in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION_REG	GPPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0

The clock source is selected by the T0CS bit:

T0CS = 0 selects timer mode, where TMR0 is incremented at a fixed rate by the instruction clock.

T0CS = 1 selects counter mode, where TMR0 is incremented by an external signal, on the T0CKI pin. On the PIC12F629, this is physically the same pin as GP2.

T0CKI is a Schmitt Trigger input, meaning that it can be driven by and will respond cleanly to a smoothly varying input voltage (e.g. a sine wave), even with a low level of superimposed noise; it doesn't have to be a sharply defined TTL-level signal, as required by the GP inputs.

In counter mode, the T0SE bit selects whether Timer0 responds to rising or falling signals ("edges") on T0CKI. Clearing T0SE to '0' selects the rising edge; setting T0SE to '1' selects the falling edge.

Prescaler

By default, the timer increments by one for every instruction cycle (in timer mode) or transition on T0CKI (in counter mode). If timer mode is selected, and the processor is clocked at 4 MHz, the timer will increment at the instruction cycle rate of 1 MHz. That is, TMR0 will increment every 1 μ s. Thus, with a 4 MHz clock, the maximum period that Timer0 can measure directly, by default, is 255 μ s.

To measure longer periods, we need to use the *prescaler*.

The prescaler sits between the clock source and the timer. It is used to reduce the clock rate seen by the timer, by dividing it by a power of two: 2, 4, 8, 16, 32, 64, 128 or 256.

To use the prescaler with Timer0, clear the PSA bit to '0'¹.

When assigned to Timer0, the prescale ratio is set by the PS<2:0> bits, as shown in the following table:

PS<2:0> bit value	Timer0 prescale ratio
000	1 : 2
001	1 : 4
010	1 : 8
011	1 : 16
100	1 : 32
101	1 : 64
110	1 : 128
111	1 : 256

If PSA = 0 (assigning the prescaler to Timer0) and PS<2:0> = '111' (selecting a ratio of 1:256), TMR0 will increment every 256 instruction cycles in timer mode. Given a 1 MHz instruction cycle rate, the timer would increment every 256 μ s.

Thus, when using the prescaler with a 4 MHz processor clock, the maximum period that Timer0 can measure directly is $255 \times 256 \mu\text{s} = 65.28\text{ms}$.

Note that the prescaler can also be used in counter mode, in which case it divides the external signal on T0CKI by the prescale ratio.

If you don't want to use the prescaler with Timer0, for a 1:1 "prescale ratio", set PSA to '1'.

¹ If PSA = 1, the prescaler is assigned to the watchdog timer – a topic covered in [lesson 7](#).

Timer Mode

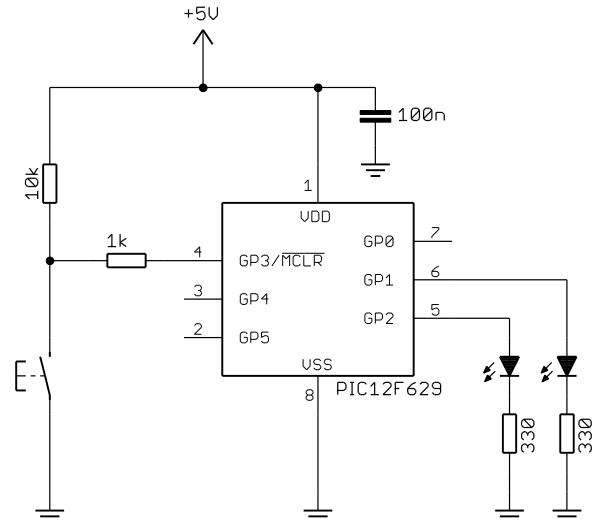
The examples in this section demonstrate the use of Timer0 in timer mode, to:

- Measure elapsed time
- Perform a regular task while responding to user input
- Debounce a switch

For each of these, we'll use the circuit shown on the right, which adds an LED to the circuit used in [lesson 3](#). A second LED has been added to GP2, although any of the unused pins would have been suitable.

If you have the [Gooligum training board](#), connect jumpers JP3, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP1 and GP2.

If you are using Microchip's Low Pin Count Demo Board, you will need to connect LEDs to GP1 and GP2, as described in [baseline lesson 1](#).



Example 1: Reaction Timer

To illustrate how Timer0 can be used to measure elapsed time, we'll implement a very simple reaction time "game": wait a couple of seconds then light an LED to indicate 'start'. If the button is pressed within a predefined time (say 200 ms) light the other LED to indicate 'success'. If the user is too slow, leave the 'success' LED unlit. Either way, delay another second before turning off the LEDs and restarting.

There are many enhancements we could add, to make this a better game. For example, success/fail could be indicated by a bi-colour red/green LED. The delay prior to the 'start' indication should be random, so that it's difficult to cheat by predicting when it's going to turn on. The difficulty level could be made adjustable, and the measured reaction time in milliseconds could be displayed, using 7-segment displays. You can probably think of more – but the intent of here is to keep it as simple as possible, while providing a real-world example of using Timer0 to measure elapsed time.

We'll use the LED on GP2 as the 'start' signal and the LED on GP1 to indicate 'success'.

The program flow can be illustrated in pseudo-code as:

```
do forever
  clear both LEDs
  delay 2 sec
  indicate start
  clear timer
  wait up to 1 sec for button press
  if button pressed and elapsed time < 200 ms
    indicate success
  delay 1 sec
end
```

A problem is immediately apparent: even with maximum prescaling, Timer0 can only measure up to 65 ms. To overcome this, we need to extend the range of the timer by adding a counter variable, which is incremented when the timer overflows. That means monitoring the value in TMR0 and incrementing the counter variable when TMR0 reaches a certain value.

This example utilises the (nominally) 4 MHz internal RC clock, giving an instruction cycle time of (approximately) 1 μ s. Using the prescaler, with a ratio of 1:32, means that the timer increments every 32 μ s. If we clear TMR0 and then wait until TMR0 = 250, 8 ms (250 \times 32 μ s) will have elapsed. If we then reset TMR0 and increment a counter variable, we've implemented a counter which increments every 8 ms. Since 25 \times 8 ms = 200 ms, when the counter reaches 25, 200 ms will have elapsed; any counter value > 25 means that the allowed time has been exceeded. And since 125 \times 8 ms = 1 s, when the counter reaches 125, one second will have elapsed and we can stop waiting for the button press.

The following code sets Timer0 to timer mode (internal clock, freeing GP2 to be used as an output), with the prescaler assigned to Timer0, with a 1:32 prescale ratio:

```

    movlw    b'11000100'    ; configure Timer0:
                        ; --0-----    timer mode (T0CS = 0)
                        ; ----0----    prescaler assigned to Timer0 (PSA = 0)
                        ; -----100    prescale = 32 (PS = 100)
    banksel  OPTION_REG    ; -> increment TMR0 every 32 us
    movwf   OPTION_REG

```

This code is setting bits 6 and 7 of OPTION_REG, even though these bits ($\overline{\text{GPPU}}$ and INTEDG) are not related to Timer0. In the baseline architecture, there is no choice but to load the whole of the OPTION register at once, but for mid-range PICs it is possible to use bit set/clear instructions to modify individual bits in OPTION_REG, or to use logical *masking* operations to update only some bit fields, leaving other bits unchanged.

For example, to preserve the contents of OPTION_REG<6:7>, you could write:

```

    banksel  OPTION_REG
    movf     OPTION_REG,w    ; operate on OPTION_REG
    andlw   b'11000000'    ; while preserving bits 6-7
    iorlw   b'00000100'
                        ; --0-----    timer mode (T0CS = 0)
                        ; ----0----    prescaler assigned to Timer0 (PSA = 0)
                        ; -----100    prescale = 32 (PS = 100)
    movwf   OPTION_REG    ; -> increment TMR0 every 32 us

```

The 'andlw' and 'iorlw' instructions respectively perform "logical and" and "inclusive-or" operations on the W register with the given literal (constant) value, placing the result in W – "and literal with W" and "inclusive-or literal with W".

However, given that, by default (after a power-on reset), every bit in OPTION_REG is set to '1', there is no real need to go to the trouble to use masks to preserve bits 6 and 7; we know that they were already set to '1'. Nevertheless, in some cases you will want to update only part of a register, so it's worth taking the time to understand how these masking operations work. There will be more examples in later lessons.

Assuming a 4 MHz clock, such as the internal RC oscillator, TMR0 will begin incrementing every 32 μ s.

To generate an 8 ms delay, we can clear TMR0 and then wait until it reaches 250, as follows:

```

    banksel  TMR0          ; clear Timer0
    clrf    TMR0
w_tmr0    movf    TMR0,w    ; wait for 8 ms
          xorlw   .250      ; (250 ticks x 32 us/tick = 8 ms)
          btfss  STATUS,Z
          goto   w_tmr0

```

Note that XOR is used to test for equality (TMR0 = 250), as we did in [lesson 3](#).

In itself, that's an elegant way to create a delay; it's much shorter and simpler than "busy loops", such as the delay routines from lessons [1](#) and [2](#). But the real advantage of using a timer is that it keeps ticking over, at the same rate, while other instructions are executed. That means that additional instructions can be inserted

into this “timer wait” loop, without affecting the timing – within reason; if this extra code takes too long to run, the timer may increment more than once before it is checked at the end of the loop, and the loop may not finish when intended.

However long the additional code is, it takes some time to run, so the timer increment will not be detected immediately. This means that the overall delay will be a little longer than intended. For that reason (and others), it is usually better to use timer-driven interrupts for tasks like this, as we will see in [lesson 6](#).

That’s not a problem in this example, where exact timing is not important, so with 32 instruction cycles per timer increment, it’s safe to insert a short piece of code to check whether the pushbutton has been checked.

For example:

```

        banksel TMR0           ; clear Timer0
        clrf   TMR0
w_tmr0  ; repeat for 8 ms:
        banksel GPIO
        btfss GPIO,GP3       ; if button pressed (GP3 low)
        goto  wait_end       ; finish delay loop immediately
        banksel TMR0
        movf  TMR0,w         ;
        xorlw .250           ; (250 ticks x 32 us/tick = 8 ms)
        btfss STATUS,Z
        goto  w_tmr0
wait_end

```

This timer loop code can then be embedded into an outer loop which increments a variable used to count the number of 8 ms periods, as follows:

```

        clrf   cnt_8ms       ; clear timer (8 ms counter)
wait1s  ; repeat for 1 sec:
        banksel TMR0
        clrf   TMR0         ; clear Timer0
w_tmr0  ; repeat for 8 ms:
        banksel GPIO
        btfss GPIO,3       ; if button pressed (GP3 low)
        goto  wait1s_end   ; finish delay loop immediately
        banksel TMR0
        movf  TMR0,w       ;
        xorlw .250         ; (250 ticks x 32 us/tick = 8 ms)
        btfss STATUS,Z
        goto  w_tmr0
        incf  cnt_8ms,f    ; increment 8 ms counter
        movlw .125        ; (125 x 8 ms = 1 sec)
        xorwf cnt_8ms,w
        btfss STATUS,Z
        goto  wait1s
wait1s_end

```

The test at the end of the outer loop ($\text{cnt_8ms} = 125$) ensures that the loop completes when 1 s has elapsed, in case the button has not been pressed.

Finally, we need to check whether the user has pressed the button quickly enough (if at all). That means comparing the elapsed time, as measured by the 8 ms counter, with some threshold value – in this case 25, corresponding to a reaction time of 200 ms. The user has been successful if the 8 ms count is less than 25.

The easiest way to compare the magnitude of two values (is one larger or smaller than the other?) is to subtract them, and see if a *borrow* results.

If $A \geq B$, $A - B$ is positive or zero and no borrow is needed.

If $A < B$, $A - B$ is negative, requiring a borrow.

Mid-range PICs have two subtraction instructions:

‘subw_f f, d’ – “**subtract W from file register**”, where ‘f’ is the register and, ‘d’ is the destination;
 ‘, f’ to write the result back to the register: $f = f - W$
 ‘, w’ to place the result in W: $W = f - W$

and:

‘sublw k’ – “**subtract W from literal**”, where ‘k’ is the literal value to subtract W from;
 the result is placed in W: $W = k - W$

Note that there is no instruction which subtracts a literal from W. Or is there?

Recall that the expression ‘W - k’ is equivalent to ‘W + (-k)’, i.e. adding a negative value is equivalent to subtracting a positive value.

We saw in [baseline lesson 11](#) that when negative values are represented in *two’s complement* format, the normal binary integer addition and subtraction operations continue to work, in a consistent way, with both positive and negative numbers.

The ‘addlw’ instruction is used to **add** a literal to **W**.

The ‘-’ operator is used by the MPASM assembler to specify a two’s complement value, so to subtract a literal from W, we can simply write:

‘addlw -k’, which performs the operation: $W = W - k$

Whichever way the subtraction is performed, the result is reflected in the Z (zero) and C (carry) bits in the STATUS register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	IRP	RP1	RP0	\overline{TO}	\overline{PD}	Z	DC	C

The Z bit is set if and only if the result is zero (so subtraction is another way to test for equality).

Although the C bit is called “carry”, in a subtraction it acts as a “not borrow”. That is, it is set to ‘1’ only if a borrow did *not* occur.

The table at the right shows the possible status flag outcomes from the subtraction A - B:

	Z	C
A > B	0	1
A = B	1	1
A < B	0	0

We can make use of this to test whether the elapsed time is less than 200 ms (`cnt_8ms < 25`) as follows:

```

movlw    .25                ; if time < 200 ms (25 x 8 ms)
subwf    cnt_8ms,w          ; (cnt_8ms < 25)
banksel  GPIO
btfss    STATUS,C
bsf      GPIO,GP1          ; turn on success LED
    
```

The subtraction performed here is `cnt_8ms - 25`, so **C** = 0 only if `cnt_8ms < 25` (see the table above).

If **C** = 1, the elapsed time must be greater than the allowed 200 ms, and the instruction to turn on the success LED is skipped.

Note that the 'banksel GPIO' directive is placed above the 'btfss' instruction. This is important. If we had instead written this as:

```
btfss    STATUS,C
banksel  GPIO
bsf      GPIO,GP1          ; turn on success LED
```

the instruction generated by `banksel2` is skipped if `C` is set, instead of the `bcf` instruction.

That is not at all what was intended; keep in mind that the 'banksel' directive generates instructions which are inserted into your code, so sometimes (as in this example) you need to be careful where you place it, to avoid unexpected side-effects.

Note also that there is never any need to use `banksel` before accessing the `STATUS` register, because it is mapped into the same address in every bank.

Alternatively, we could use the `sublw` instruction to perform the comparison:

```
btn_dn  movf    cnt_8ms,w          ; if time < 200 ms (25 x 8 ms)
        sublw   .24                ; (cnt_8ms <= 24)
        banksel GPIO
        btfsc   STATUS,C
        bsf     GPIO,GP1          ; turn on success LED
```

Note that the sense of the subtraction performed here ($24 - \text{cnt_8ms}$) is reversed from the one above.

According to the truth table on the previous page, we now have to test for `C = 1` instead of `C = 0` and the comparison becomes '`≤`' instead of '`<`', meaning that the comparison has to be with 24 instead of 25.

Or, we could even use `addlw` for the subtraction (comparison):

```
movf    cnt_8ms,w          ; if time < 200 ms (25 x 8 ms)
addlw   -.25              ; (cnt_8ms < 25)
banksel GPIO
btfss   STATUS,C
bsf     GPIO,GP1          ; turn on success LED
```

Complete program

Here's the complete code for the reaction timer, using the 'subwf'-based comparison routine:

```
*****
; Description:      Lesson 4, example 1a                               *
;                  Reaction Timer game.                             *
;                                                          *
; Demonstrates use of timer0 to time real-world events           *
;                                                          *
; User must attempt to press button within 200 ms of "start" LED *
; lighting.  If and only if successful, "success" LED is lit.    *
;                                                          *
; Starts with both LEDs unlit.                                    *
; 2 sec delay before lighting "start"                            *
; Waits up to 1 sec for button press                             *
; (only) on button press, lights "success"                       *
; 1 sec delay before repeating from start                        *
;                                                          *
; (version using subwf instruction in comparison routine)        *
```

² On mid-range PICs with four register banks, `banksel` generates two instructions; only the first will be skipped.

```

;*****
;
; Pin assignments:
; GP1 = success LED
; GP2 = start LED
; GP3 = pushbutton switch (active low)
;
;*****

list      p=12F629
#include   <p12F629.inc>

errorlevel -302    ; no "register not in bank 0" warnings
errorlevel -312    ; no "page or bank selection not needed" messages

EXTERN    delay10      ; W x 10 ms delay

;***** CONFIGURATION
                ; int reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
__CONFIG      _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

;***** VARIABLE DEFINITIONS
                UDATA_SHR
cnt_8ms res 1      ; counter: increments every 8 ms

;***** RESET VECTOR *****
RESET CODE 0x0000    ; processor reset vector
                ; calibrate internal RC oscillator
call 0x03FF        ; retrieve factory calibration value
banksel OSCCAL     ; (stored at 0x3FF as a retlw k)
movwf OSCCAL       ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start
                ; configure port
movlw ~(1<<GP1|1<<GP2) ; configure GP1 and GP2 (only) as outputs
banksel TRISIO
movwf TRISIO

                ; configure timer
movlw b'11000100'    ; configure Timer0:
                ; --0----- timer mode (T0CS = 0)
                ; ----0--- prescaler assigned to Timer0 (PSA = 0)
                ; -----100 prescale = 32 (PS = 100)
banksel OPTION_REG ; -> increment TMR0 every 32 us
movwf OPTION_REG

;***** Main loop
main_loop
                ; turn off both LEDs
banksel GPIO
clrf GPIO

```

```

; delay 2 sec
movlw    .200                ; 200 x 10 ms = 2 sec
pagesel delay10
call     delay10
pagesel $

; indicate start
banksel GPIO
bsf      GPIO,GP2           ; turn on start LED

; wait up to 1 sec for button press
wait1s   clrf    cnt_8ms      ; clear timer (8 ms counter)
          ; repeat for 1 sec:
banksel TMR0
wait1s   clrf    TMR0        ; clear Timer0
          ; repeat for 8 ms:
w_tmr0   banksel GPIO
          btfs   GPIO,3      ; if button pressed (GP3 low)
          goto   wait1s_end  ; finish delay loop immediately
banksel TMR0
w_tmr0   movf    TMR0,w      ;
          xorlw  .250        ; (250 ticks x 32 us/tick = 8 ms)
          btfs   STATUS,Z
          goto   w_tmr0
          incf   cnt_8ms,f    ; increment 8 ms counter
          movlw  .125        ; (125 x 8 ms = 1 sec)
          xorwf  cnt_8ms,w
          btfs   STATUS,Z
          goto   wait1s
wait1s_end

; indicate success if elapsed time < 200 ms
movlw    .25                ; if time < 200 ms (25 x 8 ms)
subwf   cnt_8ms,w          ; (cnt_8ms < 25)
banksel GPIO
btfs   STATUS,C
bsf     GPIO,GP1          ; turn on success LED

; delay 1 sec
movlw    .100               ; 100 x 10 ms = 1 sec
pagesel delay10
call     delay10
pagesel $

; repeat forever
goto    main_loop

END

```

Example 2: Flash LED while responding to input

As discussed above, timers can be used to maintain the accurate timing of regular (“background”) events, while performing other actions in response to input signals. To illustrate this, we’ll flash the LED on GP2 at 1 Hz (similar to the second example in [lesson 1](#)), while lighting the LED on GP1 whenever the pushbutton on GP3 is pressed (as was done in [lesson 3](#)).

We’ll see in [lesson 6](#) that timer-driven interrupts are ideally suited to performing regular background tasks. This example is only included here for completeness; it’s not how you would implement this, on a mid-range PIC, in practice.

When creating an application which performs a number of tasks, it is best, if practical, to implement and test each of those tasks separately. In other words, build the application a piece at a time, adding each new part to base that is known to be working. So we'll start by simply flashing the LED.

The delay needs to be written in such a way that button scanning code can be added within it later. Calling a delay subroutine, as was done in [lesson 2](#), wouldn't be appropriate; if the button press was only checked at the start and/or end of the delay, the button would seem unresponsive (a 0.5 sec delay is very noticeable).

Since the maximum delay that Timer0 can produce directly from a 1 MHz instruction clock is 65 ms, we have to extend the timer by adding a counter variable, as was done in example 1.

To produce a given delay, various combinations of prescaler value, maximum timer count and number of repetitions will be possible. But noting that $125 \times 125 \times 32 \mu\text{s} = 500 \text{ ms}$, a delay of exactly 500 ms can be generated by:

- Using a 4 MHz processor clock, giving a 1 MHz instruction clock and a 1 μs instruction cycle
- Assigning a 1:32 prescaler to the instruction clock, incrementing Timer0 every 32 μs
- Resetting Timer0 to zero, as soon as it reaches 125 (i.e. every $125 \times 32 \mu\text{s} = 4 \text{ ms}$)
- Repeating 125 times, creating a delay of $125 \times 4 \text{ ms} = 500 \text{ ms}$.

The following code implements the above steps:

```

;***** Initialisation
start
    ; configure port
    banksel GPIO
    clrf GPIO ; start with all LEDs off
    clrf sGPIO ; update shadow
    movlw ~(1<<GP1|1<<GP2) ; configure GP1 and GP2 (only) as outputs
    banksel TRISIO ; (GP3 is an input)
    movwf TRISIO

    ; configure timer
    movlw b'11000100' ; configure Timer0:
    ; --0----- timer mode (T0CS = 0)
    ; ----0--- prescaler assigned to Timer0 (PSA = 0)
    ; -----100 prescale = 32 (PS = 100)
    banksel OPTION_REG ; -> increment TMR0 every 32 us
    movwf OPTION_REG

;***** Main loop
main_loop
    ; delay 500 ms
    movlw .125 ; repeat 125 times (125 x 4 ms = 500 ms)
    movwf dly_cnt

dly500
    banksel TMR0 ; clear timer0
    clrf TMR0
w_tmr0 movf TMR0,w ; wait for 4 ms
    xorlw .125 ; (125 ticks x 32 us/tick = 4 ms)
    btfss STATUS,Z
    goto w_tmr0
    decfsz dly_cnt,f ; end 500 ms delay loop
    goto dly500

    ; toggle flashing LED
    movf sGPIO,w

```



```

xorlw    1<<GP2           ; toggle LED on GP2
movwf   sGPIO            ; using shadow register
banksel GPIO
movwf   GPIO

; repeat forever
goto   main_loop

```

Note that, strictly speaking, the ‘banksel’ directives within the main loop are not needed, because the only registers accessed within the loop, TMR0 and GPIO, are in the same bank. Nevertheless, it’s good practice to include these directives, as shown, in case you later insert some code which changes the bank selection. That’s not difficult to deal with, but it’s easy to miss a situation where banksel is needed, ending up with a difficult-to-find bug. If you use banksel liberally, even when not strictly needed, your code will be a little longer, but much more easily maintained.

Here’s the code developed in [lesson 3](#), for turning on an LED when the pushbutton is pressed:

```

clrf    sGPIO            ; assume button up -> LED off
btfss  GPIO,GP3         ; if button pressed (GP3 low)
bsf    sGPIO,GP1        ; turn on LED

movf    sGPIO,w         ; copy shadow to GPIO
movwf   GPIO

```

It’s quite straightforward to place some code similar to this (replacing the clrf with a bcf instruction, to avoid affecting any other bits in the shadow register) within the timer wait loop – since the timer increments every 32 instructions, there are plenty of cycles available to accommodate these additional instructions, without risk that the “TMR0 = 125” condition will be skipped (see discussion in example 1).

Here’s how:

```

w_tmr0           ; repeat for 4 ms:
banksel GPIO     ; check and respond to button press
bcf    sGPIO,GP1 ; assume button up -> indicator LED off
btfss  GPIO,GP3 ; if button pressed (GP3 low)
bsf    sGPIO,GP1 ; turn on indicator LED
movf    sGPIO,w  ; update port (copy shadow to GPIO)
movwf   GPIO
banksel TMR0
movf    TMR0,w
xorlw   .125     ; (125 ticks x 32 us/tick = 4 ms)
btfss  STATUS,Z
goto   w_tmr0

```

Complete program

Here’s the complete code for the flash + pushbutton demo.

Note that, because GPIO is being updated from the shadow copy, every “spin” of the timer wait loop, there is no need to update GPIO when the LED on GP2 is toggled; the change will be picked up next time through the loop.

```

;*****
;
; Description: Lesson 4 example 2
;
; Demonstrates use of Timer0 to maintain timing of background tasks
; while performing other actions in response to changing inputs
;

```

```

; One LED simply flashes at 1 Hz (50% duty cycle). *
; The other LED is only lit when the pushbutton is pressed *
; *
;*****
; *
; Pin assignments: *
; GP1 = "button pressed" indicator LED *
; GP2 = flashing LED *
; GP3 = pushbutton switch (active low) *
; *
;*****

list      p=12F629
#include   <p12F629.inc>

errorlevel -302          ; no "register not in bank 0" warnings

;***** CONFIGURATION
                ; int reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
__CONFIG      _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

;***** VARIABLE DEFINITIONS
                UDATA_SHR
sGPIO        res 1          ; shadow copy of GPIO
dly_cnt      res 1          ; delay counter

;***** RESET VECTOR *****
RESET        CODE    0x0000      ; processor reset vector
                ; calibrate internal RC oscillator
                call    0x03FF      ; retrieve factory calibration value
                banksel OSCCAL      ; (stored at 0x3FF as a retlw k)
                movwf   OSCCAL      ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start
                ; configure port
                banksel GPIO
                clrf    GPIO          ; start with all LEDs off
                clrf    sGPIO        ; update shadow
                movlw   ~(1<<GP1)    ; configure GP1 (only) as output
                banksel TRISIO       ; (GP3 is an input)
                movwf   TRISIO

                ; configure timer
                movlw   b'11000100'  ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----100        prescale = 32 (PS = 100)
                banksel OPTION_REG    ; -> increment TMR0 every 32 us
                movwf   OPTION_REG

;***** Main loop

```

```

main_loop
    ; delay 500 ms while responding to button press
    movlw    .125                ; repeat 125 times (125 x 4 ms = 500 ms)
    movwf   dly_cnt

dly500
    banksel TMR0                ; clear timer0
    clrf    TMR0

w_tmr0
    banksel GPIO                ; check and respond to button press
    bcf     sGPIO,GP1           ; assume button up -> indicator LED off
    btfss  GPIO,GP3            ; if button pressed (GP3 low)
    bsf     sGPIO,GP1           ; turn on indicator LED
    movf    sGPIO,w             ; update port (copy shadow to GPIO)
    movwf   GPIO
    banksel TMR0
    movf    TMR0,w
    xorlw   .125                ; (125 ticks x 32 us/tick = 4 ms)
    btfss  STATUS,Z
    goto    w_tmr0
    decfsz dly_cnt,f           ; end 500 ms delay loop
    goto    dly500

    ; toggle flashing LED
    movf    sGPIO,w
    xorlw   1<<GP2             ; toggle LED on GP2
    movwf   sGPIO              ; using shadow register
    banksel GPIO
    movwf   GPIO

    ; repeat forever
    goto    main_loop

END

```

Example 3: Switch debouncing

[Lesson 3](#) explored the topic of switch bounce, and described a counting algorithm to address it, which was expressed as:

```

count = 0
while count < max_samples
    delay sample_time
    if input = required_state
        count = count + 1
    else
        count = 0
end

```

The switch is deemed to have changed when it has been continuously in the new state for some minimum period, for example 10 ms. This is determined by continuing to increment a count while checking the state of the switch. “Continuing to increment a count” while something else occurs, such as checking a switch, is exactly what a timer does. Since a timer increments automatically, using a timer can simplify the logic, as follows:

```

reset timer
while timer < debounce time
    if input ≠ required_state
        reset timer
end

```

On completion, the input will have been in the required state (changed) for the minimum debounce time.

Assuming a 1 MHz instruction clock and a 1:64 prescaler, a 10 ms debounce time will be reached when the timer reaches $10\text{ ms} \div 64\ \mu\text{s} = 156.3$; taking the next highest integer gives 157.

The following code demonstrates how Timer0 can be used to debounce a “button down” event:

```

        banksel TMR0
wait_dn clrf    TMR0           ; reset timer
chk_dn  btfsc  GPIO,GP3      ; check for button press (GP3 low)
        goto   wait_dn       ; continue to reset timer until button down
        movf   TMR0,w        ; has 10 ms debounce time elapsed?
        xorlw  .157          ; (157 = 10ms/64us)
        btfss STATUS,Z       ; if not, continue checking button
        goto   chk_dn

```

That’s shorter than the equivalent routine presented in [lesson 3](#), and it avoids the need to use two data registers as counters. But – it uses Timer0. Although mid-range PICs have more than one timer, they are still a scarce resource. You must be careful, as you build a library of routines that use Timer0, that if you use more than one routine which uses Timer0 in a single program, that the way they use or setup Timer0 doesn’t clash. As we’ll see in [lesson 6](#), it can be better to use a regular timer-driven interrupt for switch debouncing, allowing a single timer (driving the interrupt) to be used for a number of tasks.

But if you’re not using Timer0 for anything else, using it for switch debouncing is perfectly reasonable.

Complete program

The following program is equivalent to that presented in [lesson 3](#):

```

;*****
;
; Description: Lesson 4, example 3
;
; Demonstrates use of Timer0 to implement debounce counting algorithm
;
; Toggles LED when pushbutton is pressed then released
;
;*****
;
; Pin assignments:
; GP1 = LED
; GP3 = pushbutton switch (active low)
;
;*****

list      p=12F629
#include   <p12F629.inc>

errorlevel -302           ; no "register not in bank 0" warnings

;***** CONFIGURATION
                ; int reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
__CONFIG      _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

;***** VARIABLE DEFINITIONS
                UDATA_SHR
sGPIO        res 1           ; shadow copy of GPIO

```

```

;***** RESET VECTOR *****
RESET    CODE    0x0000        ; processor reset vector
        ; calibrate internal RC oscillator
        call    0x03FF        ; retrieve factory calibration value
        banksel OSCCAL        ; (stored at 0x3FF as a retlw k)
        movwf   OSCCAL        ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start
        ; configure port
        banksel GPIO
        clrf    GPIO          ; start with all LEDs off
        clrf    sGPIO         ; update shadow
        movlw   ~(1<<GP1|1<<GP2) ; configure GP1 and GP2 (only) as outputs
        banksel TRISIO        ; (GP3 is an input)
        movwf   TRISIO

        ; configure timer
        movlw   b'11000101'    ; configure Timer0:
        ; --0-----        timer mode (T0CS = 0)
        ; ----0---         prescaler assigned to Timer0 (PSA = 0)
        ; -----101       prescale = 64 (PS = 101)
        banksel OPTION_REG    ; -> increment TMR0 every 64 us
        movwf   OPTION_REG

;***** Main loop
main_loop
        ; wait for button press, debounce using timer0:
        banksel TMR0
wait_dn  clrf    TMR0          ; reset timer
chk_dn   btfs   GPIO,GP3     ; check for button press (GP3 low)
        goto    wait_dn      ; continue to reset timer until button down
        movf    TMR0,w       ; has 10 ms debounce time elapsed?
        xorlw   .157         ; (157 = 10ms/64us)
        btfs   STATUS,Z      ; if not, continue checking button
        goto    chk_dn

        ; toggle LED on GP1
        banksel GPIO
        movf    sGPIO,w
        xorlw   1<<GP1        ; toggle shadow register
        movwf   sGPIO
        movwf   GPIO         ; write to port

        ; wait for button release, debounce using timer0:
        banksel TMR0
wait_up  clrf    TMR0          ; reset timer
chk_up   btfs   GPIO,GP3     ; check for button release (GP3 high)
        goto    wait_up      ; continue to reset timer until button up
        movf    TMR0,w       ; has 10 ms debounce time elapsed?
        xorlw   .157         ; (157 = 10ms/64us)
        btfs   STATUS,Z      ; if not, continue checking button
        goto    chk_up

        ; repeat forever
        goto    main_loop

END

```

Counter Mode

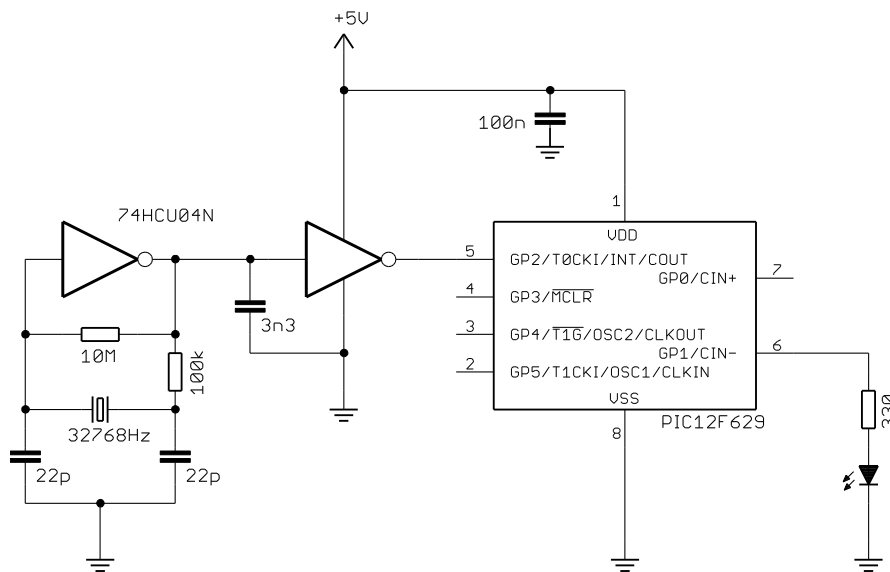
As mentioned above, Timer0 can also be used to count transitions (rising or falling) on the T0CKI input.

This is useful in a number of ways, such as performing an action after some number of events, or measuring the frequency of an input signal, for example from a sensor triggered by the rotation of an axle. The frequency in Hertz of the signal is simply the number of transitions counted in one second.

However, it's not really practical to build a frequency counter, using only the techniques (and microcontrollers) we've covered so far!

To illustrate the use of Timer0 as a counter, we'll go back to LED flashing, but driving the counter with a crystal-based external clock, providing a much more accurate time base.

The circuit used for this is shown below (with the reset switch and pull-up omitted for clarity).



An oscillator based on a 32.768 kHz “watch crystal” and a CMOS inverter was presented in [baseline lesson 5](#). It is used again here to generate a 32.768 kHz clock signal, which drives the 12F629’s T0CKI input, via an inverting buffer.

The [Gooligum training board](#) already has this oscillator circuit in place (in the upper right of the board) – close jumper JP22 to connect the 32 kHz clock signal to T0CKI.

And, as before, close

jumpers JP3 and JP12 to enable the external $\overline{\text{MCLR}}$ pull-up resistor (not shown here) and the LED on GP1.

If you have Microchip’s Low Pin Count Demo Board, you will need to build the oscillator circuit separately and connect it to the 14-pin header on the demo board (GP2/T0CKI is brought out as pin 9 on the header, while power and ground are pins 13 and 14), as shown in [baseline lesson 5](#).

We’ll use this clock input to generate the timing needed to flash the LED on GP1 at almost exactly 1 Hz (the accuracy being set by the accuracy of the crystal oscillator, which can be expected to be much better than that of the PIC’s internal RC oscillator).

Those familiar with binary numbers will have noticed that $32768 = 2^{15}$, making it very straightforward to divide the 32768 Hz input down to 1 Hz.

Since $32768 = 128 \times 256$, if we apply a 1:128 prescale ratio to the 32768 Hz signal on T0CKI, TMR0 will be incremented 256 times per second. The most significant bit of TMR0 (TMR0<7>) will therefore be cycling at a rate of exactly 1 Hz; it will be ‘0’ for 0.5 s, followed by ‘1’ for 0.5 s.

So if we clock TMR0 with the 32768 Hz signal on T0CKI, prescaled by 128, the task is simply to light the LED (GP1 high) when TMR0<7> = 1, and turn off the LED (GP1 low) when TMR0<7> = 0.

To configure Timer0 for counter mode (external clock on TOCKI) with a 1:128 prescale ratio, set the T0CS bit to '1', PSA to '0' and PS<2:0> to '110':

```

movlw    b'11110110'    ; configure Timer0:
          ; --1-----    counter mode (T0CS = 1)
          ; ----0----    prescaler assigned to Timer0 (PSA = 0)
          ; -----110    prescale = 128 (PS = 110)
banksel  OPTION_REG     ; -> increment at 256 Hz with 32.768 kHz input
movwf   OPTION_REG

```

Note that the value of TOSE bit is irrelevant; we don't care if the counter increments on the rising or falling edge of the signal on TOCKI – only the frequency is important. Either edge will do.

Next we need to continually set GP1 high whenever TMR0<7> = 1, and low whenever TMR0<7> = 0.

In other words, continually update GP1 with the current value of TMR0<7>.

Unfortunately, there is no simple “copy a single bit” instruction in mid-range PIC assembler!

If you're not using a shadow register for GPIO, the following “direct approach” is effective, if a little inelegant:

```

loop     ; transfer TMR0<7> to GP1
banksel  TMR0
btfsc   TMR0,7          ; if TMR0<7>=1
bsf     GPIO,GP1        ; set GP1
btfss   TMR0,7          ; if TMR0<7>=0
bcf     GPIO,GP1        ; clear GP1

        ; repeat forever
goto    loop

```

As we saw in [lesson 3](#), if you are using a shadow register (generally a good idea...), this can be implemented as:

```

loop     ; transfer TMR0<7> to GP1
clr     sGPIO           ; assume TMR0<7>=0 -> LED off
banksel  TMR0
btfsc   TMR0,7          ; if TMR0<7>=1
bsf     sGPIO,GP1       ; turn on LED

movf    sGPIO,w         ; copy shadow to GPIO
banksel  GPIO
movwf   GPIO

        ; repeat forever
goto    loop

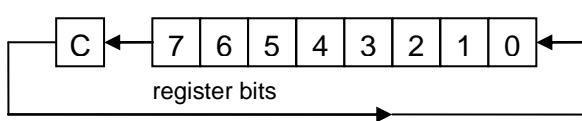
```

But since this is actually an instruction longer, it's only really simpler if you were going to use a shadow register anyway.

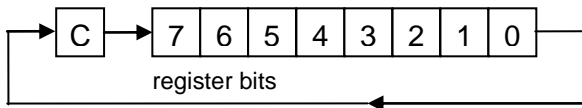
And note that the use of a single 'banksel' directive at the start of the first routine³, but two 'banksel's in the second. This is because, in a real program, where a shadow register is being used, it is likely to be updated a number of times before being copied to GPIO at the end of the loop; additional code within the loop may alter the bank selection.

³ We can get away with this because TMR0 and GPIO are in the same bank, and it's very unlikely that we'd want to insert additional code into this small routine in future – so having selected the correct bank for TMR0, we can safely access GPIO within the same routine.

Another approach is to use the PIC's rotate instructions. These instructions move every bit in a register to the left or right, as illustrated:



'rlf f,d' – "rotate left file register through carry"



'rrf f,d' – "rotate right file register through carry"

In both cases, the bit being rotated out of bit 7 (for rlf) or bit 0 (for rrf) is copied into the carry bit in the STATUS register, and the previous value of carry is rotated into bit 0 (for rlf) or bit 7 (for rrf).

As usual, 'f' is the register being rotated, and 'd' is the destination: 'f' to write the result back to the register, or 'w' to place the result in W.

The ability to place the result in W is useful, since it means that we can "left rotate" TMR0, to copy the current value from TMR0<7> into C, without affecting the value in TMR0.

In the mid-range architecture, only the special-function and general purpose registers can be rotated; there are no instructions for rotating W. That's a pity, since such an instruction would be useful here.

Instead, we must rotate the bit copied from TMR0<7> into bit 0 of a temporary register, then another rotate to move the copied bit into bit 1, and then copy the result to GPIO, as follows:

```

rlf    TMR0,w           ; copy TMR0<7> to C
clrf   temp
rlf    temp,f           ; rotate C into temp
rlf    temp,w           ; rotate once more into W (-> W<1> = TMR0<7>)
movwf  GPIO             ; update GPIO with result (-> GP1 = TMR0<7>)

```

Note that 'temp' is cleared before being used. That's not strictly necessary in this example; since the only output is GP1, it doesn't matter what the other bits in GPIO are set to.

Of course, if any other bits in GPIO were being used as outputs, you couldn't use this method, since this code will clear every bit other than GP1! In that case, you're better off using the bit test and set/clear instructions, which are generally the most practical way to "copy a bit". But it's worth remembering that the rotate instructions are also available, and using them may lead to shorter code.

Complete program

Here's the complete "flash an LED at 1 Hz using a crystal oscillator" program, using the "copy a bit via rotation" method:

```

;*****
; Description: Lesson 4, example 4b *
; *
; Demonstrates use of Timer0 in counter mode *
; *
; LED flashes at 1 Hz (50% duty cycle), *
; with timing derived from 32.768 kHz input on T0CKI *
; *
; Uses rotate instructions to copy MSB from Timer0 to GP1 *
;*****
; Pin assignments: *
; GP1 = flashing LED *
; T0CKI = 32.768 kHz signal *
;*****

```



```

list          p=12F629
#include      <p12F629.inc>

errorlevel   -302                ; no "register not in bank 0" warnings

;***** CONFIGURATION
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
__CONFIG     _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

;***** VARIABLE DEFINITIONS
                UDATA_SHR
temp         res 1                ; temp register used for rotates

;***** RESET VECTOR *****
RESET       CODE    0x0000        ; processor reset vector
                ; calibrate internal RC oscillator
                call    0x03FF        ; retrieve factory calibration value
                banksel OSCCAL        ; (stored at 0x3FF as a retlw k)
                movwf  OSCCAL        ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start
                ; configure port
                movlw   ~(1<<GP1)    ; configure GP1 (only) as an output
                banksel TRISIO
                movwf  TRISIO

                ; configure timer
                movlw   b'11110110'  ; configure Timer0:
                ; --1-----        counter mode (T0CS = 1)
                ; ----0---         prescaler assigned to Timer0 (PSA = 0)
                ; -----110       prescale = 128 (PS = 110)
                banksel OPTION_REG    ; -> increment at 256 Hz with 32.768 kHz input
                movwf  OPTION_REG

;***** Main loop
main_loop
                ; TMR0<7> cycles at 1 Hz, so continually copy to LED (GP1)
                banksel TMR0
                rlf    TMR0,w        ; copy TMR0<7> to C
                clrf  temp
                rlf    temp,f        ; rotate C into temp
                rlf    temp,w        ; rotate once more into W (-> W<1> = TMR0<7>)
                movwf GPIO          ; update GPIO with result (-> GP1 = TMR0<7>)

                ; repeat forever
                goto   main_loop

END

```

Conclusion

Hopefully the examples in this lesson have given you an idea of the flexibility and usefulness of the Timer0 peripheral.

With it, we were able to:

- Time an event
- Perform a periodic action while responding to input
- Debounce a switch
- Count external pulses

However, as mentioned a few times now, one of the most useful applications of timers is to drive interrupts, which are arguably the most significant enhancement in the mid-range architecture, and the topic of [lesson 6](#).

But first, in the [next lesson](#) we'll take a quick look at some of the features of the MPASM assembler, which can make your code easier to maintain.

Introduction to PIC Programming

Mid-range Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 5: Assembler Directives and Macros

As the programs presented in these tutorials become longer, it's time to look at some of the facilities that MPASM (the Microchip PIC assembler) provides to simplify the process of writing and maintaining code.

This lesson repeats the material from [baseline lesson 6](#), updated for the 12F629. If you have read that lesson, you can skip this one; MPASM provides the same features for baseline and mid-range PICs.

This lesson covers:

- Arithmetic and bitwise operators
- Text substitution with `#define`
- Defining constants with `equ` or `constant`
- Conditional assembly using `if/else/endif`, `ifdef` and `ifndef`
- Outputting warning and error messages
- Assembler macros

Each of these topics is illustrated by making use of it in code from previous lessons in this series.

Arithmetic Operators

MPASM supports the following arithmetic operators:

negate	-
multiply	*
divide	/
modulus	%
add	+
subtract	-

Precedence is in the traditional order, as above.

For example, $2 + 3 * 4 = 2 + 12 = 14$.

To change the order of precedence, use parentheses: (and) .

For example, $(2 + 3) * 4 = 5 * 4 = 20$.

*Note: These calculations take place during the assembly process, before any code is generated. They are used to calculate constant values which will be included in the code to be assembled. They **do not** generate any PIC instructions.*

These arithmetic operators are useful in showing how a value has been derived, making it easier to understand the code and to make changes.

For example, consider this code from [lesson 1](#):

```

; delay 500 ms
movlw    .244           ; outer loop: 244 x (1023 + 1023 + 3) + 2
movwf    dc2           ;   = 499,958 cycles
clrf     dc1           ; inner loop: 256 x 4 - 1
dly1     nop           ; inner loop 1 = 1023 cycles
         decfsz    dc1,f
         goto      dly1
dly2     nop           ; inner loop 2 = 1023 cycles
         decfsz    dc1,f
         goto      dly2
         decfsz    dc2,f
         goto      dly1

```

Where does the value of 244 come from? It is the number of outer loop iterations needed to make 500 ms.

To make this clearer, we could change the comments to:

```

; delay 500 ms
movlw    .244           ; outer loop: #iterations =
movwf    dc2           ;   500ms/(1023+1023+3)us/loop = 244

```

Or, instead of writing the constant '244' directly, write it as an expression:

```

; delay 500 ms
movlw    .500000/(.1023+.1023+.3) ; number of outer loop iterations
movwf    dc2                 ; for 500 ms delay

```

If you're using mainly decimal values in your expressions, as here, you may wish to change the default radix to decimal, to avoid having to add a '.' before each decimal value. As discussed in [lesson 1](#), that's not necessarily a good idea; if your code assumes that some particular default radix has been set, you need to be very careful if you copy that code into another program, which may have a different default radix. But, if you're prepared to take the risk, add the 'radix' directive near the start of the program. For example:

```
radix    dec
```

The valid radix values are 'hex' for hexadecimal (base 16), 'dec' for decimal (base 10) and 'oct' for octal (base 8). The default radix is hex.

With the default radix set to decimal, this code fragment can be written as:

```

; delay 500 ms
movlw    500000/(1023+1023+3) ; # outer loop iterations for 500 ms
movwf    dc2

```

Defining Constants

Programs often contain numeric values which may need to be tuned or changed later, particularly during development. When a change needs to be made, finding these values in the code can be difficult. And making changes may be error-prone if the same value (or another value derived from the value being changed) occurs more than once in the code.

To make the code more maintainable, each constant value should be defined only once, near the start of the program, where it is easy to find and change.

A good example is the reaction timer developed in [lesson 4](#), where “success” was defined as pressing a pushbutton less than 200 ms after a LED was lit. But what if, during testing, we found that 200 ms is unrealistically short? Or too long?

To change this maximum reaction time, you’d need to find and then modify this fragment of code:

```

; indicate success if elapsed time < 200 ms
movlw  .25                ; if time < 200 ms (25 x 8 ms)
subwf  cnt_8ms,w         ; (cnt_8ms < 25)
banksel GPIO
btfss  STATUS,C
bsf    GPIO,GP1         ; turn on success LED

```

To make this easier to maintain, we could define the maximum reaction time as a constant, at the start of the program.

This can be done using the ‘`equ`’ (short for “equate”) directive, as follows:

```
MAXRT  equ    .200        ; Maximum reaction time in ms
```

Alternatively, you could use the ‘`constant`’ directive:

```
constant MAXRT=.200      ; Maximum reaction time in ms
```

The two directives are equivalent. Which you choose to use is simply a matter of style.

‘`equ`’ is more commonly found in assemblers, and perhaps because it is more familiar, most people use it.

Personally, I prefer to use ‘`constant`’, mainly because I like to think of any symbol placed on the left hand edge (column 1) of the assembler source as being a label for a program or data register address, and I prefer to differentiate between address labels and constants to be used in expressions. But it’s purely your choice.

However you define this constant, it can be referred to later in your code, for example:

```

; check elapsed time
movlw  MAXRT/8           ; if time < max reaction time (8 ms/count)
subwf  cnt_8ms,w
banksel GPIO
btfss  STATUS,C
bsf    GPIO,GP1         ; turn on success LED

```

Note how constants can be usefully included in arithmetic expressions. In this way, the constant can be defined simply in terms of real-world quantities (e.g. ms), making it readily apparent how to change it to a new value (e.g. 300 ms), while arithmetic expressions are used to convert that into a quantity that matches the program’s logic. And if that logic changes later (say, counting by 16 ms instead of 8 ms increments), then only the arithmetic expression needs to change; the constant can remain defined in the same way.

And of course, since [lesson 1](#), we’ve been using constants defined in the processor include file, such as ‘`GP1`’, in instructions such as:

```
bsf    GPIO,GP1         ; turn on success LED
```

Text Substitution

As discussed above, the ability to define numeric constants is very useful. It is also very useful to be able to define “text constants”, where a text *string* is substituted into the assembler source code.

Text substitution is commonly used to refer to I/O pins by a descriptive label. This makes your code more readable, and easier to update if pin assignments change later.

Why would pin assignments change? Whether you design your own printed circuit boards, or layout your circuit on prototyping board, swapping pins around can often simplify the physical circuit layout. That's one of the great advantages of designing with microcontrollers; as you layout your design, you can go back and modify the code to simplify that layout, perhaps repeating that process a number of times.

For example, consider again the reaction timer from [lesson 4](#). The I/O pins were assigned as follows:

```
; Pin assignments:                                     *
; GP1 = success LED                                   *
; GP2 = start LED                                     *
; GP3 = pushbutton                                    *
```

These assignments are completely arbitrary; the LEDs could be on any pin other than GP3 (which is input only), while the pushbutton could be on any unused pin.

One way of defining these pins would be to use numeric constants:

```
constant nSTART=2           ; Start LED
constant nSUCCESS=1        ; Success LED
constant nBUTTON=3         ; pushbutton
```

(The 'n' prefix used here indicates that these are numeric constants; this is simply a convention, and you can choose whatever naming style works for you.)

They would then be referenced in the code, as follows:

```
        bsf      GPIO,nSTART           ; turn on start LED
w_tmr0  btfss   GPIO,nBUTTON          ; check for button press (low)
        bsf      GPIO,nSUCCESS        ; turn on success LED
```

A significant problem with this approach is that larger PICs (i.e. most of them!) have more than one port. Instead of GPIO, larger PICs have ports named PORTA, PORTB, PORTC and so on. What if you moved an input or output from PORTA to PORTC? The above approach, using numeric constants, wouldn't work, because you'd have to go through your code and change all the PORTA references to PORTC.

This problem can be solved using text substitution, using the '#define' directive, as follows:

```
; pin assignments
#define START      GPIO,2           ; LEDs
#define SUCCESS    GPIO,1
#define BUTTON     GPIO,3           ; pushbutton
```

These definitions are then referenced later in the code, as shown:

```
        bsf      START                 ; turn on start LED
w_tmr0  btfss   BUTTON                ; check for button press (low)
        bsf      SUCCESS               ; turn on success LED
```

Note that there are no longer any references to GPIO in the main body of the code. If you later move this code to a PIC with more ports, you only need to update the definitions at the start. Of course, you also need to modify the corresponding port initialisation code, such loading the TRIS registers, normally located at the start of the program, or in an "init" subroutine.

Bitwise Operators

We've seen that operations on binary values are fundamental to PIC microcontrollers: setting and clearing individual bits, flipping bits, testing the status of bits and rotating the bits in registers. It is common to have to specify individual bits, or combinations of bits, when loading values into registers, such as `TRISIO` or `OPTION_REG`, or using directives such as `'__CONFIG'`.

To facilitate operations on bits, MPASM provides the following bitwise operators:

compliment	~
left shift	<<
right shift	>>
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	

Precedence is in the order listed above.

As with arithmetic operators, parentheses are used to change the order of precedence: `'()'`.

We've seen an example of the bitwise AND operator in every program so far:

```
__CONFIG    _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF &
            _WDT_OFF & _PWRTE_ON & _INTRC_OSC_NOCLKOUT
```

These symbols are defined in the `'p12F629.inc'` include file as follows:

```
----- CONFIG Options -----
_LP_OSC      EQU  H'3FF8'    ; LP oscillator
_XT_OSC      EQU  H'3FF9'    ; XT oscillator
_HS_OSC      EQU  H'3FFA'    ; HS oscillator
_EC_OSC      EQU  H'3FFB'    ; EC
_INTRC_OSC_NOCLKOUT EQU  H'3FFC'    ; INTOSC oscillator: I/O on GP4
_INTRC_OSC_CLKOUT  EQU  H'3FFD'    ; INTOSC oscillator: CLKOUT on GP4
_EXTRC_OSC_NOCLKOUT EQU  H'3FFE'    ; RC oscillator: I/O on GP4
_EXTRC_OSC_CLKOUT  EQU  H'3FFF'    ; RC oscillator: CLKOUT on GP4
_WDT_OFF     EQU  H'3FF7'    ; WDT disabled
_WDT_ON      EQU  H'3FFF'    ; WDT enabled
_PWRTE_ON    EQU  H'3FEF'    ; PWRT enabled
_PWRTE_OFF   EQU  H'3FFF'    ; PWRT disabled
_MCLRE_OFF   EQU  H'3FDF'    ; GP3/MCLR pin function is digital I/O
_MCLRE_ON    EQU  H'3FFF'    ; GP3/MCLR pin function is MCLR
_BODEN_OFF   EQU  H'3FBF'    ; BOD disabled
_BODEN_ON    EQU  H'3FFF'    ; BOD enabled
_CP_ON       EQU  H'3F7F'    ; Program Memory code protection enabled
_CP_OFF      EQU  H'3FFF'    ; Program Memory code protection disabled
_CPD_ON      EQU  H'3EFF'    ; Data memory code protection enabled
_CPD_OFF     EQU  H'3FFF'    ; Data memory code protection disabled
```

The `'equ'` directive is described above; you can see that these are simply symbols for numeric constants.

In binary, the values in the `'__CONFIG'` directive above are:

```
_MCLRE_OFF   H'3FDF' = 11 1111 1101 1111
_CP_OFF      H'3FFF' = 11 1111 1111 1111
_CPD_OFF     H'3FFF' = 11 1111 1111 1111
_BODEN_OFF   H'3FBF' = 11 1111 1011 1111
_WDT_OFF     H'3FF7' = 11 1111 1111 0111
_PWRTE_ON    H'3FEF' = 11 1111 1110 1111
_INTRC_OSC_NOCLKOUT H'3FFC' = 11 1111 1111 1100
```

ANDing these together gives: 11 1111 1000 0100

So the directive above is equivalent to:

```
__CONFIG    b'11111110000100'
```

For each of these configuration bit symbols, where a bit in the definition is '0', it has the effect of setting the corresponding bit in the configuration word to '0', because a '0' ANDed with '0' or '1' always equals '0'.

The 14-bit configuration word in the PIC12F629 is as shown:

Bit 13	12	11	10	9	8	7	6	5	4	3	2	1	Bit 0
BG1	BG0	-	-	-	$\overline{\text{CPD}}$	$\overline{\text{CP}}$	BODEN	MCLRE	$\overline{\text{PRWTE}}$	WDTE	FOSC2	FOSC1	FOSC0

Most of these configuration options were described briefly in [lesson 1](#). Recapping:

$\overline{\text{CPD}}$ disables data memory code protection. Clearing $\overline{\text{CPD}}$ prevents the contents of the EEPROM from being read externally (your PIC program can still read the EEPROM, whatever $\overline{\text{CPD}}$ is set to).

$\overline{\text{CP}}$ disables program memory code protection. Clearing $\overline{\text{CP}}$ protects your program code from being read by PIC programmers.

BODEN enables brown-out detection, resetting the PIC if the supply voltage drops below a preset level, to increase system reliability.

MCLRE enables the external processor reset, or “master clear” ($\overline{\text{MCLR}}$), on pin 4. Clearing it allows GP3 to be used as an input.

$\overline{\text{PWRT}}\overline{\text{E}}$ disables the power-up timer, which holds the device in reset for approximately 72 ms after power is first applied, or after the supply voltage recovers following a brown-out, to allow the power supply to stabilise.

The BG bits are used to calibrate the “bandgap” voltage, used as an internal reference for brown-out detection and power-on reset (when power is first applied, the PIC is not released from reset until a sufficiently high supply voltage, related to the bandgap reference, is reached). These BG<1:0> bits are programmed in the factory and are normally preserved when the PIC is programmed.

WDTE enables the watchdog timer, which is used to reset the processor if it crashes, as we’ll see in a later lesson. Clearing WDTE to disables the watchdog timer.

The FOSC bits set the clock, or oscillator, configuration; FOSC<2:0> = 100 specifies the internal RC oscillator with no clock output. The other oscillator configurations will be described in a later lesson.

Given this, to configure the PIC12F629 for internal reset (GP3 as an input), no code or data protection, no watchdog timer, no brownout detection, with the power-up timer enabled and using the internal RC oscillator with no clock output, the lower nine bits of the configuration word must be set to: 110000100.

That’s the same pattern of bits as produced by the __CONFIG directive, above (the value of bits 9 to 11 is irrelevant, as they are not used, and bits 12 and 13 are factory-set), showing that deriving the individual bit settings from the data sheet gives the same result as using the symbols in the Microchip-provided include file – as it should! But using the symbols is simpler, and safer; it’s easy to mistype a long binary value, leading to a difficult-to-debug processor configuration error. If you mistype a symbol, the assembler will tell you, making it easy to correct the mistake.

As discussed in [lesson 1](#), it is also useful to be able to use symbols instead of binary numbers when setting bits in special-function registers, such as TRISIO or OPTION_REG.

The bits in the OPTION register are defined in the 'p12F629.inc' include file as follows:

```

;----- OPTION_REG Bits -----
PSA          EQU  H'0003'
T0SE         EQU  H'0004'
T0CS         EQU  H'0005'
INTEDG       EQU  H'0006'
NOT_GPPU     EQU  H'0007'
PS0          EQU  H'0000'
PS1          EQU  H'0001'
PS2          EQU  H'0002'

```

These are different to the symbol definitions used for the configuration bits, as they define a bit *position*, not a pattern.

It tells us, for example, that T0CS is bit 5. Having these symbols defined make it possible to write, for example:

```
bsf    OPTION_REG,T0CS    ; select counter mode: T0CS=1
```

Using symbols in this way makes the code clearer, and it is harder to make a mistake, as mistyping a symbol is likely to be picked up by the assembler, while mistyping a numeric constant (such as writing "bsf OPTION_REG, 4" when the intention was to set bit 5, or T0CS) is more likely to be missed.

Typically a number of bits in a single register need to be configured at the same time. To do this in a single instruction, using symbols, it is possible to use the bitwise operators to build expressions referencing a number of symbols; something we have been doing to load the TRISIO register, since [lesson 1](#).

For example, the "flash led while responding to pushbutton" code from [lesson 4](#) included:

```

movlw   ~(1<<GP1|1<<GP2)    ; configure GP1 and GP2 as outputs
banksel TRISIO              ; (GP3 is an input)
movwf   TRISIO

```

This makes use of the compliment, left-shift and inclusive-OR operators to build an expression equivalent to the binary constant '1111001', which is loaded into TRISIO.

Sometimes it makes sense to leave a bit field, such as PS<2:0> in OPTION_REG, expressed as a binary constant, while using symbols to set or clear other, individual, bits in the register.

For example, the crystal-based LED flasher code from [lesson 4](#) included:

```

movlw   b'11110110'        ; configure Timer0:
; --1-----              counter mode (T0CS = 1)
; ----0----              prescaler assigned to Timer0 (PSA = 0)
; -----110             prescale = 128 (PS = 110)
banksel OPTION_REG        ; -> increment at 256 Hz with 32.768 kHz input
movwf   OPTION_REG

```

This can be rewritten as:

```

movlw   1<<T0CS|0<<PSA|b'110'
; counter mode (T0CS = 1)
; prescaler assigned to Timer0 (PSA = 0)
; prescale = 128 (PS = 110)
banksel OPTION_REG        ; -> increment at 256 Hz with 32.768 kHz input
movwf   OPTION_REG

```

Including '0<<PSA' in the expression does nothing, since a zero left-shifted any number of times is still zero, and ORing zero into any expression has no effect. But it makes it explicit that we are clearing PSA.

Since we don't care what the $\overline{\text{GPPU}}$, INTEDG and TOSE bits are set to, they are not included in the expression.

Macros

We saw in [lesson 2](#) that, if we wish to reuse the same piece of code a number of times in a program, it often makes sense to place that code into a subroutine and to call the subroutine from the main program.

But that's not always appropriate, or even possible. The subroutine call and return is an overhead that takes some time; only four instruction cycles, but in timing-critical pieces of code, it may not be justifiable. And although mid-range PICs have an eight-level deep stack (compared with only two levels in the baseline architecture), you must still be careful when nesting subroutine calls, or else the stack will overflow and your subroutine won't return to the right place. It may not be worth using up a stack level, just to avoid repeating a short piece of code.

Another problem with subroutines is that, as we saw in lesson 2, to pass parameters to them, you need to load the parameters into registers – an overhead that leads to longer code, perhaps negating the space-saving advantage of using a subroutine, for small pieces of code. And loading parameters into registers, before calling a subroutine, isn't very readable. It would be nicer to be able to simply list the parameters on a single line, as part of the subroutine call.

Macros address these problems, and are often appropriate where a subroutine is not. A *macro* is a sequence of instructions that is inserted (or *expanded*) into the source code by the assembler, prior to assembly.

*Note: The purpose of a macro is to make the source code more compact; unlike a subroutine, it **does not** make the resultant object code any smaller. The instructions within a macro are expanded into the source code, every time the macro is called.*

Here's a simple example. [Lesson 2](#) introduced a 'delay10' subroutine, which took as a parameter in W a number of multiples of 10 ms to delay. So to delay for 200 ms, we had:

```
movlw    .20                ; delay 20 x 10 ms = 200 ms
call     delay10
```

This was used in a program which flashed an LED with a 20% duty cycle: on for 200 ms, then off for 800 ms. Rewritten a little from the code presented in lesson 2, the main loop looks like this:

```
main_loop
    bsf     FLASH            ; turn on LED
    movlw   .20              ; stay on for 200 ms
    pagesel delay10         ; (delay 20 x 10 ms)
    call    delay10
    bcf     FLASH            ; turn off LED
    movlw   .80              ; stay off for 800 ms
    call    delay10         ; (delay 80 x 10 ms)
    pagesel $                ; repeat forever
    goto   main_loop
```

It would be nice to be able to simply write something like 'DelayMS 200' for a 200 ms delay. We can do that by defining a macro, as follows:

```
DelayMS MACRO    ms                ; delay time in ms
    movlw   ms/.10                ; divide by 10 to pass to delay10 routine
    pagesel delay10
    call    delay10
    pagesel $
ENDM
```

This defines a macro called 'DelayMS', which takes a single parameter: 'ms', the delay time in milliseconds. Parameters are referred to within the macro in the same way as any other symbol, and can be used in expressions, as shown.

A macro definition consists of a label (the macro's name), the 'MACRO' directive, and a comma-separated list of symbols, or *arguments*, used to pass parameters to the macro, all on one line.

It is followed by a sequence of instructions and/or assembler directives, finishing with the 'ENDM' directive.

When the source code is assembled, the macro's instruction sequence is inserted into the code, with the arguments replaced by the parameters that were passed to the macro.

That may sound complex, but using a macro is easy. Having defined the 'DelayMS' macro, as above, it can be called from the main loop, as follows:

```
main_loop
    bsf     FLASH           ; turn on LED
    DelayMS .200           ; stay on for 200ms
    bcf     FLASH           ; turn off LED
    DelayMS .800           ; stay off for 800ms
    goto   main_loop       ; repeat forever
```

This 'DelayMS' macro is a *wrapper*, making the 'delay10' subroutine easier to use.

Note that the `pagesel` directives have been included as part of the macro, first to select the correct page for the 'delay10' subroutine, and then to select the current page again after the subroutine call. That makes the macro transparent to use; there is no need for `pagesel` directives before or after calling it.

As a more complex example, consider the debounce code presented in [lesson 4](#):

```
wait_dn clrf    TMR0           ; reset timer
chk_dn  btfsc   GPIO,GP3      ; check for button press (GP3 low)
        goto   wait_dn       ; continue to reset timer until button down
        movf   TMR0,w         ; has 10 ms debounce time elapsed?
        xorlw  .157           ; (157 = 10ms/64us)
        btfss  STATUS,Z      ; if not, continue checking button
        goto   chk_dn
```

If you had a number of buttons to debounce in your application, you would want to use code very similar to this, multiple times. But since there is no way of passing a reference to the pin to debounce (such as 'GPIO, GP3') as a parameter to a subroutine, you would need to use a macro to achieve this.

For example, a debounce macro could be defined as follows:

```
; Debounce switch on given input port,pin
; Waits for switch to be 'high' continuously for 10 ms
;
; Uses:      TMR0           Assumes: TMR0 running at 256 us/tick
;
DbnceHi MACRO port,pin
    local    start,wait,DEBOUNCE
    variable DEBOUNCE=.10*.1000/.256 ; debounce count = 10ms/(256us/tick)

    pagesel $           ; select current page for gotos
    banksel TMR0        ; and correct bank for TMR0 and port
start  clrf    TMR0     ; button down so reset timer (counts "up" time)
wait   btfss  port,pin ; wait for switch to go high (=1)
        goto   start
        movf   TMR0,w   ; has switch has been up continuously for
        xorlw  DEBOUNCE ; debounce time?
        btfss  STATUS,Z ; if not, keep checking that it is still up
        goto   wait
    ENDM
```

There are a few things to note about this macro definition, starting with the comments. As with subroutines, you'll eventually build up a library of useful macros, which you might keep together in an include file, such as 'stdmacros.inc' (which you would reference using the #include directive, instead of copying the macros into your code.) When documenting a macro, it's important to note any resources (such as timers) used by the macro, and any initialisation that has to have been done before the macro is called.

The macro is called 'DbnceHi' instead of 'DbnceUp' because it's waiting for a pin to be consistently high. For some switches, that will correspond to "up", but not in every case. Using terms such as "high" instead of "up" is more general, and thus more reusable.

The 'local' directive declares symbols (address labels and variables) which are only used within the macro. If you call a macro more than once, you must declare any address labels within the macro as "local", or else the assembler will complain that you have used the same label more than once. Declaring macro labels as local also means that you don't need to worry about whether those labels are used within the main body of code. A good example is 'start' in the definition above. There is a good chance that there will be a 'start' label in the main program, but that doesn't matter, as the *scope* of a label declared to be "local" is limited to the macro it is defined in.

The 'variable' directive is very similar to the 'constant' directive, introduced earlier. The only difference is that the symbol it defines can be updated later. Unlike a constant, the value of a variable can be changed after it has been defined. Other than that, they can be used interchangeably.

In this case, the symbol 'DEBOUNCE' is being defined as a variable, but is used as a constant. It is never updated, being used to make it easy to change the debounce period from 10 ms if required, without having to find the relevant instruction within the body of the macro (and note the way that an arithmetic expression has been used, to make it easy to see how to set the debounce to some other number of milliseconds).

So why define 'DEBOUNCE' as a variable, instead of a constant? If it was defined as a constant, there would potentially be a conflict if there was another constant called 'DEBOUNCE' defined somewhere else in the program. But surely declaring it to be "local" would avoid that problem? Unfortunately, the 'local' directive only applies to labels and variables, not constants. And that's why 'DEBOUNCE' is declared as a "local variable". Its scope is limited to the macro and will not affect anything outside it. You can't do that with constants.

Finally, note that the macro begins with a 'pagesel \$' directive. That is placed there because we cannot assume that the page selection bits are set to the current page when the macro is called. If the current page was not selected, the 'goto' commands within the macro body would fail; they would jump to a different page. That illustrates another difference between macros and subroutines: when a subroutine is called, the page the subroutine is on must have been selected (or else it couldn't have been called successfully), so any 'goto' commands within the subroutine will work. You can't safely make that assumption for macros. Similarly a 'banksel TMRO' is included, since we cannot be sure that, when the macro is called, the correct bank for accessing TMRO has been selected. Note also that, because TMRO and all of the port registers are in bank 0 for every mid-range PIC, this will also select the correct bank for whichever port is being used.

Complete program

The following program demonstrates how this "debounce" macro is used in practice.

It is based on the "toggle an LED" program included in [lesson 4](#), but the press of the pushbutton is not debounced, only the release. It is not normally necessary to debounce both actions – although you may have to think about it a little to see why!

Using the macro doesn't make the code any shorter, but the main loop is much simpler:

```
;*****
; Description:      Lesson 5, example 4                *
;                  Toggles LED when button is pressed *
;                  *                                  *
; Demonstrates use of macro defining Timer0-based debounce routine *
;*****
```

```

; Pin assignments:
; GP1 = LED
; GP3 = pushbutton switch (active low)
;
;*****

list      p=12F629
#include  <p12F629.inc>

errorlevel -302      ; no "register not in bank 0" warnings
errorlevel -312      ; no "page or bank selection not needed" messages

;***** CONFIGURATION
                ; int reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
__CONFIG      _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

; pin assignments
constant      nLED=1                ; indicator LED on GP1
#define       BUTTON GPIO,3          ; pushbutton on GP3

;***** MACROS
; Debounce switch on given input port,pin
; Waits for switch to be 'high' continuously for 10 ms
;
; Uses:        TMR0                Assumes: TMR0 running at 256 us/tick
;
DbnceHi MACRO  port,pin
    local      start,wait,DEBOUNCE
    variable   DEBOUNCE=.10*.1000/.256 ; switch debounce count =
10ms/(256us/tick)

                pagesel $                ; select current page for gotos
                banksel TMR0             ; and correct bank for TMR0 and port
start  clrf    TMR0                       ; button down so reset timer (counts "up" time)
wait   btfss   port,pin                   ; wait for switch to go high (=1)
        goto   start
        movf   TMR0,w                     ; has switch has been up continuously for
        xorlw  DEBOUNCE                   ; debounce time?
        btfss  STATUS,Z                   ; if not, keep checking that it is still up
        goto   wait
        ENDM

;***** VARIABLE DEFINITIONS
                UDATA_SHR
sGPIO   res 1                               ; shadow copy of GPIO

;***** RESET VECTOR *****
RESET   CODE    0x0000                     ; processor reset vector
                ; calibrate internal RC oscillator
        call   0x03FF                     ; retrieve factory calibration value
        banksel OSCCAL                    ; (stored at 0x3FF as a retlw k)
        movwf  OSCCAL                      ; then update OSCCAL

;***** MAIN PROGRAM *****

```

```

;***** Initialisation
start
    ; configure port
    banksel GPIO
    clrf GPIO ; start with all LEDs off
    clrf sGPIO ; update shadow
    movlw ~(1<<nLED) ; configure LED pin (only) as output
    banksel TRISIO ; (GP3 is an input)
    movwf TRISIO

    ; configure timer (for DbnceHi macro)
    movlw b'11000111' ; configure Timer0:
    ; --0----- timer mode (T0CS = 0)
    ; ----0--- prescaler assigned to Timer0 (PSA = 0)
    ; -----111 prescale = 256 (PS = 111)
    banksel OPTION_REG ; -> increment TMR0 every 256 us
    movwf OPTION_REG

;***** Main loop
main_loop
    ; wait for button press
    banksel GPIO
wait_dn btfsc BUTTON ; wait until button low
    goto wait_dn

    ; toggle LED
    movf sGPIO,w
    xorlw 1<<nLED ; toggle shadow register
    movwf sGPIO
    movwf GPIO ; write to port

    ; wait for button release
    DbnceHi BUTTON ; wait until button high (debounced)

    ; repeat forever
    goto main_loop

END

```

Conditional Assembly

We've seen how the processor include files, such as 'p12F629.inc', define a number of symbols that allow you to refer to registers and flags by name, instead of numeric value.

While looking at the 'p12F629.inc' file, you may have noticed these lines:

```

IFNDEF __12F629
    MESSG "Processor-header file mismatch. Verify selected processor."
ENDIF

```

This is an example of *conditional assembly*, where the actions performed by the assembler (outputting messages and generating code) depend on whether specific conditions are met.

When the processor is specified by the 'list p=' directive, or selected in MPLAB, a symbol specifying the processor is defined; for the PIC12F629, the symbol is '__12F629'. This is useful because the assembler can be made to perform different actions depending on which processor symbol has been defined.

In this case, the idea is to check that the correct processor include file is being used. If you include the include file for the wrong processor, you'll almost certainly have problems. This code checks for that.

The `'IFDEF'` directive instructs the assembler to assemble the block of code following it if the specified symbol *has not* been defined.

The `'ENDIF'` directive marks the end of the block of conditionally-assembled code.

In this case, everything between `'IFDEF'` and `'ENDIF'` is assembled if the symbol `'__12F629'` has not been defined. And that will only be true if a processor other than the PIC12F629 has been selected.

The `'MESSG'` directive tells the assembler to print the specified message in the MPLAB output window. This message is only informational; it's useful for providing information about the assembly process or for issuing warnings that do not necessarily mean that assembly has to stop.

So, this code tests that the correct processor has been selected and, if not, warns the user about the mismatch.

Similar to `'IFDEF'`, there is also an `'IFDEF'` directive which instructs the assembler to assemble a block of code only if the specified symbol *has* been defined.

A common use of `'IFDEF'` is when debugging, perhaps to disable parts of the program while it is being debugged. Or you might want to use a different processor configuration, say with code protection and brownout detection enabled. For example:

```
#define      DEBUG

IFDEF DEBUG
                ; int reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
    __CONFIG    _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
                _PWRTE_ON & _INTRC_OSC_NOCLKOUT
ELSE
                ; int reset, code and data protect on, brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
    __CONFIG    _MCLRE_OFF & _CP_ON & _CPD_ON & _BODEN_ON & _WDT_OFF &
                _PWRTE_ON & _INTRC_OSC_NOCLKOUT
ENDIF
```

If the `'DEBUG'` symbol has been defined (it doesn't have to be set equal to anything, just defined), the first `__CONFIG` directive is assembled, turning off code protection and the watchdog timer.

The `'ELSE'` directive marks the beginning of an alternative block of code, to be assembled if the previous conditional block was not selected for assembly.

That is, if the `'DEBUG'` symbol has *not* been defined, the second `__CONFIG` directive is assembled, turning on code protection and the watchdog timer.

When you have finished debugging, you can either comment out the `'#define DEBUG'` directive, or change `'DEBUG'` to another symbol, such as `'RELEASE'`. The debugging code will now no longer be assembled.

In many cases, simply testing whether a symbol exists is not enough. You may want the assembler to assemble different sections of code and/or issue different messages, depending on the value of a symbol, or of an expression containing perhaps a number of symbols.

As an example, suppose your code is used to support a number of hardware configurations, or revisions. At some point the printed circuit board may have been revised, requiring different pin assignments. In that case, you could use a block of code similar to:

```
constant      REV='A'                ; hardware revision

; pin assignments
```

```

IF REV=='A'                                ; pin assignments for REV A:
    constant    nLED=1                      ; indicator LED on GP1
    #define     BUTTON    GPIO,3           ; pushbutton on GP3
ENDIF
IF REV=='B'                                ; pin assignments for REV B:
    constant    nLED=0                      ; indicator LED on GP0
    #define     BUTTON    GPIO,2           ; pushbutton on GP2
ENDIF
IF REV!='A' && REV!='B'
    ERROR "Revision must be 'A' or 'B'"
ENDIF

```

This code allows for two hardware revisions, selected by setting the constant 'REV' equal to 'A' or 'B'.

If you have the [Gooligum training board](#), it's easy to try this out: connect jumpers JP7 and JP3 to enable pull-up resistors on GP2 and GP3, and jumpers JP11 and JP12 to enable LEDs on GP0 and GP1.

The 'IF *expr*' directive instructs the assembler to assemble the following block of code if the expression *expr* is true. Normally a *logical expression* (such as a test for equality) is used with the 'IF' directive, but arithmetic expressions can also be used, in which case an expression that evaluates to zero is considered to be logically false, while any non-zero value is considered to be logically true.

MPASM supports the following logical operators:

not (logical compliment)	!
greater than or equal to	>=
greater than	>
less than	<
less than or equal to	<=
equal to	==
not equal to	!=
logical AND	&&
logical OR	

Precedence is in the order listed above.

And as you would expect, parentheses are used to change the order of precedence: '(' and ')'.

Note that the test for equality is two equals signs; '==', not '='.

In the code above, setting 'REV' to 'A' means that the first pair of #define directives will be executed, while setting 'REV' to 'B' executes the second pair.

But what if 'REV' was set to something other than 'A' or 'B'? Then neither set of pin assignments would be selected and the symbols 'LED' and 'BUTTON' would be left undefined. The rest of the code would not assemble correctly, so it is best to check for that error condition.

This error condition can be tested for, using the more complex logical expression:

```
REV!='A' && REV!='B'
```

Incidentally, this can be rewritten equivalently¹ as:

```
!(REV=='A' || REV=='B')
```

¹ This equivalence is known as De Morgan's theorem.

You can of course use whichever form seems clearest to you.

The `'ERROR'` directive does essentially the same thing as `'MESSG'`, but instead of printing the specified message and continuing, `'ERROR'` will make the progress bar that appears during assembly turn red, and the assembly process will halt.

The `'IF'` directive is also very useful for checking that macros have been called correctly, particularly for macros which may be reused in other programs.

For example, consider the delay macro defined earlier:

```
DelayMS MACRO    ms                ; delay time in ms
    movlw    ms/.10                ; divide by 10 to pass to delay10 routine
    pagesel delay10
    call    delay10
    pagesel $
ENDM
```

The maximum delay allowed is 2.55 s, because all registers, including `W`, are 8-bit and so can only hold numbers up to 255. If you try calling `'DelayMS'` with an argument greater than 2550, the assembler will warn you about "Argument out of range", but it will carry on anyway, using the least significant 8 bits of `'ms/.10'`. That's not a desirable behaviour. It would be better if the assembler reported an error and halted, if the macro is called with an argument that is out of range.

That can be done as follows:

```
DelayMS MACRO    ms                ; delay time in ms
    IF ms>.2550
        ERROR "Maximum delay time is 2550 ms"
    ENDIF
    movlw    ms/.10                ; divide by 10 to pass to delay10 routine
    pagesel delay10
    call    delay10
    pagesel $
ENDM
```

By testing that parameters are within allowed ranges in this way, you can make your code more robust.

Conclusion

MPASM offers many more advanced facilities that can make your life as a PIC assembler programmer easier, but that's enough for now.

The features we've seen in this lesson will help make your code easier to understand and maintain, and make you more productive, by:

- using arithmetic expressions to make it clear how numeric constants are derived
- using constants and text substitution, so make your code more readable and so that future configuration changes can be made in a single place
- using symbols and bitwise operators instead of cryptic binary constants
- creating macros to make your code shorter, more readable, and easier to re-use

Other MPASM directives will be introduced in future lessons, as appropriate.

So far we've been tracking the material covered in the [baseline lessons](#) quite closely, but in the [next lesson](#) we'll introduce one of the most significant features not found in the baseline architecture – interrupts.

Introduction to PIC Programming

Mid-Range Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 6: Introduction to Interrupts

The lessons up until now have re-visited topics covered in the [baseline assembler tutorial series](#), adapting the material to mid-range PICs, and introducing specific features of the mid-range architecture (not found in baseline PICs) where relevant. The most significant of these features, not present in the baseline architecture, is support for *interrupts*. As we will see in this lesson, interrupts make it much easier to implement regular “background” tasks (such as refreshing a multiplexed display – see for example [baseline lesson 8](#)) and allow programs to respond in a timely manner to external events, without having to sit in a *busy-wait*, or polling loop. Both of these applications of interrupts are demonstrated in this lesson.

In summary, this lesson covers:

- Introduction to interrupts on the mid-range PIC architecture
- Interrupt service routines (including saving and restoring processor context)
- Timer-driven interrupts
- Debouncing single switches with timer-driven interrupts
- External interrupts on the INT pin

Interrupts

An *interrupt* is a means of interrupting the main program flow in response to an event, so that the event can be dealt with, or *serviced*. The event (referred to an interrupt *source*) can be internal to the PIC, such as a timer overflowing, or external, such as a change on an input pin.

When the interrupt is triggered, program execution immediately jumps to an *interrupt service routine (ISR)*, which, in the mid-range PIC architecture, is always located at address 0004h. The ISR must save the current processor state, or *context* (i.e. the contents of any registers which the ISR will modify, such as W and STATUS), service the interrupt, and then restore the context before returning to the main program. In this way, the main program will never “notice” that the interrupt has happened – the interrupt will be completely transparent, except for whatever action the interrupt service routine was intended to perform.

Some examples will make this clearer! But first, some more details...

Each interrupt source can be enabled or disabled, independently.

The enable bits for the interrupt sources covered in this lesson are located in the INTCON register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INTCON	GIE	PEIE	T0IE	INTE	GPIE	T0IF	INTF	GPIF

To enable an interrupt source, that source's interrupt enable bit must be set:

TOIE enables the Timer0 interrupt, while INTE enables interrupts triggered by the external INT pin.

Interrupts are controlled overall by the global interrupt enable bit, GIE:

If GIE = 0, all interrupts are disabled.

If GIE = 1, interrupts can occur, depending on which interrupt sources are enabled.

For an interrupt to occur, that interrupt's enable bit must be set, in addition to GIE being set.

For example, to enable Timer0 interrupts, you could use:

```
movlw    1<<GIE|1<<TOIE    ; enable interrupt on Timer0 overflow
movwf   INTCON
```

Or, if you were setting up a number of interrupt sources and didn't want to allow interrupts to happen straight away, you might write something like:

```
bsf     INTCON,TOIE        ; enable Timer0 interrupt source
...
; (initialise some other things)
bsf     INTCON,GIE        ; enable interrupts
```

Context Saving

When an interrupt occurs, the current instruction completes executing, the address of the next instruction (the *return address*) is pushed onto the stack, the GIE bit is cleared to prevent any more interrupts from occurring while this interrupt is being serviced, and execution jumps to the instruction at address 0004h.

At this point, the only the program counter (PC) has been saved. Every other register holds whatever value it did when the interrupt was triggered.

As mentioned above, the ISR should be transparent to the main program. If the ISR modifies the contents of any register that the main program would "expect" to remain constant, that register should be saved at the start of the ISR, and restored to its original value returning to the main program, so that the main program will never "know" that an interrupt has occurred.

The Microchip-supplied template, '*...\MPASM Suite\Template\Object\12F629TMPO.ASM*', includes the following code which you can use as the framework of your interrupt service routine:

```
;-----
; INTERRUPT SERVICE ROUTINE
;-----

INT_VECTOR    CODE    0x0004    ; interrupt vector location
    MOVWF     W_TEMP        ; save off current W register contents
    MOVF      STATUS,w      ; move status register into W register
    MOVWF     STATUS_TEMP   ; save off contents of STATUS register

; isr code can go here or be located as a call subroutine elsewhere

    MOVF      STATUS_TEMP,w  ; retrieve copy of STATUS register
    MOVWF     STATUS        ; restore pre-isr STATUS register contents
    SWAPF    W_TEMP,f       ; restore pre-isr W register contents
    SWAPF    W_TEMP,w      ; restore pre-isr W register contents
    RETFIE                                ; return from interrupt
```

This code uses two variables, declared in the template code as:

```
INT_VAR      UDATA_SHR    0x20
W_TEMP      RES          1      ; variable used for context saving
STATUS_TEMP RES          1      ; variable used for context saving
```

to save the contents of the **W** and **STATUS** registers.

Note that these variables are placed in shared memory¹. Of course, on the PIC12F629, this is the only type of data memory available; it is all shared. But even on devices with banked as well as shared memory, it is necessary to use shared memory for context saving (at least for **W** and **STATUS**), because you cannot know which bank is selected when the interrupt is triggered. If you select a specific bank by changing the bank selection bits (**RP0** and **RP1**) in **STATUS**, you will lose the original value of these bits unless you save the contents of **STATUS** first. The only way to do that, without losing the current bank selection, is to copy **STATUS** to shared memory, before any bank selection is done. And since the only way to save the **STATUS** register is to copy it to **W** first, the current contents of **W** must be saved first.

The instructions to save the contents of **W** and **STATUS** are straightforward:

```
movwf  W_TEMP      ; save off current W register contents
movf   STATUS,w     ; move status register into W register
movwf  STATUS_TEMP ; save off contents of STATUS register
```

After this, any other registers you wish to save (such as **PCLATH**) can be copied to variables in the same way – and these variables can be in banked memory, because the bank selection bits (in **STATUS**) have been saved.

For example:

```
movwf  W_TEMP      ; save W and STATUS
movf   STATUS,w     ; to variables in shared memory
movwf  STATUS_TEMP
movf   PCLATH,w    ; save PCLATH
banksel PCLATH_TEMP ; to variable (can be in banked memory)
movwf  PCLATH_TEMP
```

To restore the context (**W**, **STATUS** and any other registers you choose to save, such as **PCLATH**) at the end of the interrupt routine, you might think that these instructions could simply be reversed:

```
banksel PCLATH_TEMP ; restore PCLATH
movf   PCLATH_TEMP,w
movwf  PCLATH
movf   STATUS_TEMP,w ; restore STATUS
movwf  STATUS
movf   W_TEMP,w     ; restore W (NO!!! This clobbers Z flag!!!)
```

Unfortunately, **this approach won't work!**

The final `movf` instruction, used above to restore the **W** register, has a side effect: it affects the **Z** flag (part of the **STATUS** register), depending on the value being copied. This means that, whatever value **Z** had before the interrupt was triggered may be lost. **Z** will be set or cleared depending on the value in the **W**, instead of retaining the value it held when the interrupt was triggered. That's almost certain to interfere with the main code – something we must avoid.

¹ The template code explicitly specifies the address (0x20) for this shared memory section. This is unnecessary; the linker can be relied on to place any data section declared by 'UDATA_SHR' correctly, within shared memory.

To restore *W* without affecting the *Z* flag, the template code employs a “trick”:

The ‘`swapf`’ instruction – “**swap** nybbles in file register” – is typically used where data is encoded in the two 4-bit *nybbles* (or “nibbles”) comprising an 8-bit byte, as we saw when discussing binary-coded decimal (BCD) in [baseline lesson 8](#). The contents of bits 0-3 of the specified register are swapped with bits 4-7.

If the swap operation is repeated, the nybbles end up back in their original order, leaving the data unchanged.

As with most instructions which operate on a register, the result of the swap operation can be written either back to the register, or to *W*. This makes it possible to use two successive `swapf` instructions to copy the original contents of a register to *W*, as in the ISR template code:

```
swapf    W_TEMP, f      ; restore pre-isr W register contents
swapf    W_TEMP, w
```

Why use this strange construct, instead of a simple ‘`movf`’?

The answer is that the `swapf` instruction does not affect any **STATUS** flags, while `movf` does. That means that it can be used to restore *W* without affecting **STATUS**, making the interrupt service routine truly transparent.

The final instruction in the ISR template is ‘`retfie`’ – “**return from interrupt and enable interrupts**”.

The `retfie` instruction pops the program counter off the stack, returning execution to the main program. It also sets the **GIE** bit, allowing interrupts to occur again.

Interrupt Flags

Given that a number of different interrupt sources may be enabled, your interrupt service routine must be able to determine which source triggered the interrupt, so that it can respond to that event.

Interrupt flags are used for this – when an interrupt event (such as a timer overflow) occurs, the corresponding interrupt flag is set, to indicate which event has occurred.

The flags for the interrupt sources covered in this lesson are also located in the **INTCON** register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INTCON	GIE	PEIE	TOIE	INTE	GPIE	TOIF	INTF	GPIF

TOIF indicates that Timer0 has overflowed, while **INTF** indicates that an external interrupt signal has been detected on the **INT** pin.

If an interrupt flag has been set, and the corresponding interrupt source is enabled, and the global interrupt enable (**GIE**) bit is also set, an interrupt will occur.

Whenever you service an interrupt, you must always clear its interrupt flag.

Whenever any interrupt event is serviced, the interrupt flag corresponding to that event must be cleared, or else the interrupt will be re-triggered immediately after interrupts are re-enabled, when the ISR exits.

Note that, whenever an event occurs, the interrupt flag for that event will be set, regardless of whether that interrupt source has been enabled.

For example, you can poll the **TOIF** flag to check to see if Timer0 has overflowed, without having to use interrupts.

Timer0 Interrupts

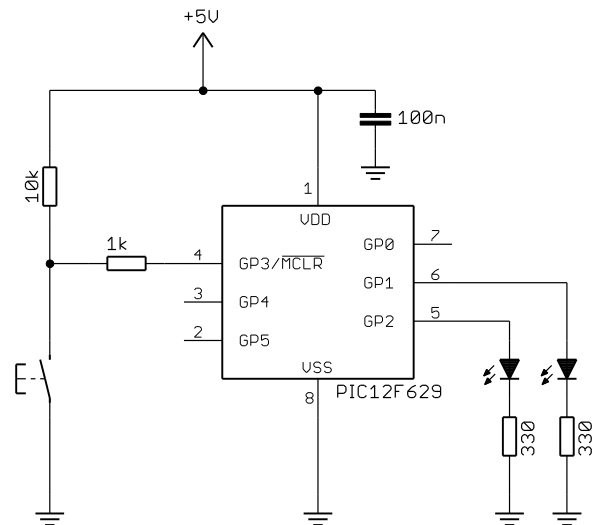
Timer0 can be used to regularly generate interrupts, which can be used to drive “background” tasks, such as:

- Generating a regular output; for example flashing an LED
- Monitoring and debouncing inputs

Meanwhile, a “main program” can continue to perform other “foreground” tasks.

We’ll use the circuit from [lesson 4](#), shown on the right, to illustrate these techniques.

If you have the [Gooligum training board](#), close jumpers JP3, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP1 and GP2.



Example 1a: Flashing an LED

To begin, we’ll simply flash an LED, without attempting to make it flash at exactly 1 Hz.

We saw in [lesson 4](#) that, given a 1 MHz instruction clock (derived from a 4 MHz processor clock) with maximum prescaling (1:256), the longest period that Timer0 can generate is $256 \times 256 \times 1 \mu\text{s} = 65.5 \text{ ms}$.

Therefore, if we configured the PIC to use a 4 MHz clock, and set up Timer0 in timer mode with a 1:256 prescaler, TMR0 would overflow (rollover from 255 to 0) every 65.5 ms.

If we then enabled Timer0 interrupts, the interrupt would be triggered on every TMR0 overflow, i.e. every 65.5 ms. So the interrupt service routine (ISR) would be called every 65.5 ms.

If the ISR toggled an LED every time it was called, the LED would change state every 65.5 ms – it would flash with a period of $65.5 \text{ ms} \times 2 = 131 \text{ ms}$, giving a frequency of 7.6 Hz.

Having an LED flash as 7.6 Hz is not ideal, but the flashing is visible (just), and that’s the slowest flash rate we can generate with the simple approach described above. So we’ll start there.

Firstly, we’ll need some variables to save the processor context during the ISR.

It’s also a good idea, when using interrupts to modify a port, to use a shadow register to avoid potential read-modify-write problems (described in [baseline lesson 2](#)). It’s usually cleaner, and safer (avoiding problems) to have only the interrupt service routine or the main program writing directly to a port (such as GPIO), but not both.

So the variable definitions we need are:

```
CONTEXT    UDATA_SHR          ; variables used for context saving
cs_W      res 1
cs_STATUS  res 1

GENVAR     UDATA_SHR          ; general variables
sGPIO     res 1               ; shadow copy of GPIO
```

Note that these could have been placed in a single section, but it’s good to get into habits that will still be appropriate for larger projects on bigger PICs; the 12F629 is a little unusual in only having a single bank of shared data memory. Giving each logical group of variables its own data section gives the linker more flexibility when allocating memory.

Since the *interrupt vector* is located at address 0004h, while the reset vector (where program execution begins) is at address 0000h, we can't continue to simply place our main program at 0000h; it could only be a maximum of four instructions long!

So it's normal to place the ISR at 0004h, and to place code at 0000h which does nothing more than jump to the start of the main program, somewhere else in memory:

```
RESET   CODE    0x0000           ; processor reset vector
        pagesel start
        goto    start

;***** INTERRUPT SERVICE ROUTINE
ISR     CODE    0x0004
        ; ISR code goes here
        ; ...
        ; end of ISR

;***** MAIN PROGRAM
MAIN    CODE
start   ; calibrate internal RC oscillator
        call   0x03FF           ; retrieve factory calibration value
        banksel OSCCAL         ; (stored at 0x3FF as a retlw k)
        movwf  OSCCAL         ; then update OSCCAL
```

Note that, because the "MAIN" code segment really could be placed anywhere in memory, it is necessary to use a 'pagesel' directive, in case the main program is located on a different page.

That won't happen on the 12F629, which only has a single page of program memory, but it's a good idea to include the 'pagesel' anyway, in case you ever move your code to a PIC with more memory.

The main program then starts by calibrating the internal RC oscillator, as shown above, before configuring and initialising the port and Timer0, as we have done before:

```
        ; configure port
        banksel GPIO
        clrf   GPIO           ; start with all LEDs off
        clrf   sGPIO         ; update shadow
        movlw  ~(1<<nLED)     ; configure LED pin (only) as an output
        banksel TRISIO
        movwf  TRISIO

        ; configure timer
        movlw  b'11000111'    ; configure Timer0:
        ; --0-----          timer mode (T0CS = 0)
        ; ----0---          prescaler assigned to Timer0 (PSA = 0)
        ; -----111        prescale = 256 (PS = 111)
        banksel OPTION_REG    ; -> increment TMR0 every 256 us
        movwf  OPTION_REG
```

There's nothing new or different here – the timer is simply set up as usual.

Now that everything is initialised and ready to go, the Timer0 interrupt can be enabled:

```
        ; enable interrupts
        movlw  1<<GIE|1<<T0IE ; enable Timer0 and global interrupts
        movwf  INTCON
```

Note that there is no need for a 'banksel' before accessing INTCON, because it is mapped into every bank.

With Timer0 setup, and the Timer0 interrupt enabled, an interrupt will be triggered every 65.5 ms, calling the interrupt service routine at 0004h.

As discussed above, the first thing the ISR must do is to save the processor context:

```
ISR      CODE      0x0004
        ; *** Save context
        movwf     cs_W           ; save W
        movf     STATUS,w       ; save STATUS
        movwf     cs_STATUS
```

We would then normally test specific interrupt flags, to determine the source of this particular interrupt. But since only Timer0 interrupts have been enabled, we know that a Timer0 overflow must have occurred, so we know that this ISR only needs to handle, or service, Timer0 overflow events.

The first thing we must do (or the last, of you prefer – it doesn't matter, as long as you ensure that you do it) is to clear the interrupt flag corresponding to this event.

In this case, because we know this is a Timer0 interrupt, we must clear TOIF:

```
        ; *** Service Timer0 interrupt
        ;
        ; TMR0 overflows every 65.5 ms
        ;
        ; Flashes LED at ~7.6 Hz by toggling on each interrupt
        ;   (every ~65.5 ms)
        ;
        ; (only Timer0 interrupts are enabled)
        ;
        bcf     INTCON,TOIF      ; clear interrupt flag
```

The interrupt routine is now free to do whatever it was intended to do; in this case, toggle an LED:

```
        ; toggle LED
        movf    sGPIO,w         ; only update shadow register
        xorlw   1<<nLED
        movwf   sGPIO
```

Note that only the shadow copy of GPIO is being updated, as discussed above.

Finally, the ISR must restore the processor context, before returning²:

```
isr_end ; *** Restore context then return
        movf    cs_STATUS,w     ; restore STATUS
        movwf   STATUS
        swapf   cs_W,f         ; restore W
        swapf   cs_W,w
        retfie
```

As mentioned earlier, the `retfie` instruction sets the GIE bit, re-enabling interrupts so that the next event (whenever it occurs) can be serviced.

² The 'isr_end' label isn't actually needed here, as it's not referenced anywhere. However, it helps to mark the end of the ISR, and it's necessary when working with multiple interrupt sources, as we'll see in the final example.

So with the ISR flipping a bit in the shadow copy of GPIO every 65.5 ms, all that remains for the main program to do is to continually copy the shadow register to the GPIO port, to make the changes made by the ISR visible (literally, in this case...):

```
main_loop
    ; continually copy shadow GPIO to port
    movf    sGPIO,w
    banksel GPIO
    movwf  GPIO

    ; repeat forever
    goto   main_loop
```

Complete program

Here is how these code fragments fit together:

```
*****
; Description: Lesson 6 example 1a *
; *
; Demonstrates use of Timer0 interrupt to perform a background task *
; *
; Flash an LED at approx 7.6 Hz (50% duty cycle). *
; *
*****
; Pin assignments: *
; GP2 = flashing LED *
; *
*****

list      p=12F629
#include   <p12F629.inc>

errorlevel -302    ; no "register not in bank 0" warnings
errorlevel -312    ; no "page or bank selection not needed" messages

;***** CONFIGURATION
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4 Mhz int clock
__CONFIG      _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

; pin assignments
constant      nLED=2                ; flashing LED on GP2

;***** VARIABLE DEFINITIONS
CONTEXT       UDATA_SHR                ; variables used for context saving
cs_W          res 1
cs_STATUS     res 1

GENVAR        UDATA_SHR                ; general variables
sGPIO         res 1                    ; shadow copy of GPIO

;***** RESET VECTOR *****
RESET        CODE    0x0000            ; processor reset vector
            pagesel start
            goto     start
```

```

;***** INTERRUPT SERVICE ROUTINE *****
ISR    CODE    0x0004
      ; *** Save context
      movwf   cs_W           ; save W
      movf    STATUS,w       ; save STATUS
      movwf   cs_STATUS

      ; *** Service Timer0 interrupt
      ;
      ; TMR0 overflows every 65.5 ms
      ;
      ; Flashes LED at ~7.6 Hz by toggling on each interrupt
      ; (every ~65.5 ms)
      ;
      ; (only Timer0 interrupts are enabled)
      ;
      bcf     INTCON,T0IF    ; clear interrupt flag

      ; toggle LED
      movf    sGPIO,w        ; only update shadow register
      xorlw   1<<nLED
      movwf   sGPIO

isr_end ; *** Restore context then return
      movf    cs_STATUS,w    ; restore STATUS
      movwf   STATUS
      swapf   cs_W,f         ; restore W
      swapf   cs_W,w
      retfie

;***** MAIN PROGRAM *****
MAIN   CODE
start  ; calibrate internal RC oscillator
      call   0x03FF         ; retrieve factory calibration value
      banksel OSCCAL        ; (stored at 0x3FF as a retlw k)
      movwf  OSCCAL         ; then update OSCCAL

;***** Initialisation

      ; configure port
      banksel GPIO
      clrf   GPIO           ; start with all LEDs off
      clrf   sGPIO          ; update shadow
      movlw  ~(1<<nLED)      ; configure LED pin (only) as an output
      banksel TRISIO
      movwf  TRISIO

      ; configure timer
      movlw  b'11000111'    ; configure Timer0:
      ; --0-----          timer mode (T0CS = 0)
      ; ----0----          prescaler assigned to Timer0 (PSA = 0)
      ; -----111         prescale = 256 (PS = 111)
      banksel OPTION_REG    ; -> increment TMR0 every 256 us
      movwf  OPTION_REG

      ; enable interrupts
      movlw  1<<GIE|1<<T0IE ; enable Timer0 and global interrupts
      movwf  INTCON

```

```

;***** Main loop
main_loop
    ; continually copy shadow GPIO to port
    movf    sGPIO,w
    banksel GPIO
    movwf   GPIO

    ; repeat forever
    goto   main_loop

END

```

Example 1b: Slower flashing

The LED in the example above flashed at around 7.6 Hz, which was done by toggling it every 65.5 ms. That's a little too fast.

We saw that, with a 4 MHz processor clock, the longest possible interval between Timer0 interrupts is 65.5 ms. So, to flash an LED any slower than this, we can't toggle it on every interrupt; we have to skip some of them. That means counting each interrupt, and only toggling the LED when the count reaches a certain value.

A simple way to implement this, if we are not concerned with exact timing, is to use an 8-bit counter, and to let it reach 255 before toggling the LED when it overflows to 0 (easily done using the `incfsz` instruction).

If, every time an interrupt is triggered by a Timer0 overflow, the ISR increments a counter, we're essentially implementing a 16-bit timer, based on Timer0, with TMR0 as the least significant eight bits, and the counter incremented by the ISR being the most significant eight bits.

If the ISR increments the counter whenever Timer0 overflows (every 256 *ticks* of TMR0), and it toggles the LED whenever the counter overflows (every 256 interrupts), the LED is being toggled every $N \times 256 \times 256$ (where N is the prescale ratio) instruction cycles.

Assuming a 1 MHz instruction clock, LED will be toggled every $N \times 256 \times 256 \mu\text{s} = N \times 65.536 \text{ ms}$.

To flash the LED at 1 Hz, we need to toggle it every 500 ms. That would require $N = 7.63$.

That's not possible, but we can use $N = 8$ (prescale ratio of 1:8), which is close – the resulting toggle period is $8 \times 256 \times 256 \mu\text{s} = 524.3 \text{ ms}$, giving a flash rate of 0.95 Hz.

That's close enough for now!

To implement the Timer0 overflow counter, we'll need a variable to store it in:

```

GENVAR    UDATA_SHR           ; general variables
sGPIO     res 1                ; shadow copy of GPIO
cnt_t0    res 1                ; counts timer0 overflows
                                   ; (incremented by ISR every 2.048 ms)

```

We then need to add instructions to the ISR to increment this counter, and toggle the LED only when it overflows back to zero:

```

    ; toggle LED every 256 interrupts (524 ms)
    incfsz cnt_t0,f           ; increment interrupt count (every 2.048 ms)
    goto   isr_end           ; if count overflow
                                   ; (every 256 interrupts = 524 ms)
    movf   sGPIO,w           ; toggle LED
    xorlw  1<<nLED           ; using shadow register
    movwf  sGPIO

```

And finally the configuration of Timer0 needs to be changed, to select a 1:8 prescaler:

```

; configure timer
movlw  b'11000010'      ; configure Timer0:
; --0-----          timer mode (T0CS = 0)
; ----0----          prescaler assigned to Timer0 (PSA = 0)
; -----010         prescale = 8 (PS = 010)
banksel OPTION_REG      ; -> increment TMR0 every 8 us
movwf  OPTION_REG

```

We should also initialise the timer overflow counter variable:

```

; initialise variables
clrf  cnt_t0           ; zero timer0 overflow count

```

although it's not strictly necessary in this example (its initial value only affects the first flash).

With these changes to the code in the first example, the LED will flash at a much more sedate 0.95 Hz.

Example 1c: Flashing an LED at exactly 1 Hz

What if we needed (for some reason) to flash the LED at exactly 1 Hz, given an accurate 4 MHz processor clock?³ Of course this is a contrived example, but there are many cases where an accurate output frequency must be generated; an obvious example is a real-time clock.

It's not possible to achieve this exact timing, using the technique in the example above, where the timer is allowed to run freely, with an interrupt being triggered every 256 ticks. Why? We need to toggle the LED every 500 ms, which, with a 4 MHz processor clock, is 500,000 instruction cycles. And 500,000 is not exactly divisible by 256 – there is no way to count to 500,000, using whole multiples of 256.

To solve this problem, we need to make the timer overflow (triggering an interrupt) every N ticks, where N divides exactly into 500,000. And, of course, since Timer0 is an 8-bit timer, $N < 256$, so it is not possible for Timer0 to count more than 256 ticks. We also want N to be as high as possible, because if Timer0 overflows less often, fewer interrupts are triggered, and less time is spent servicing interrupts.

In this case, the best result is when $N = 250$. That is, we want Timer0 to overflow after every 250 ticks.

To make a timer overflow after some number of ticks, you can preload it with an appropriate value. For example, if you had an 8-bit timer, and you wanted it to overflow after 100 ticks, you could load it with the value 156 (equal to $256 - 100$), and then start it counting. Since it is starting from 156, after 100 ticks it will have counted to 256, and overflow back to zero. The timer could then be reloaded with 156, and count for another 100 ticks, before repeating the process.

But there are some problems with this approach – some of them specific to Timer0 on mid-range PICs:

- Timer0 is always counting; there is no way to stop it incrementing, load a value, and then restart it⁴
- When a value is written to TMR0, the timer increment is inhibited for the following two instruction cycles.

³ In practice, the internal RC oscillator used in this example is only accurate to around $\pm 2\%$, varying with VDD and temperature. For higher accuracy, an external crystal should be used.

⁴ Timer0 can be halted by selecting counter mode, but then it will be incremented if there is an external signal on the T0CKI pin, so this approach is only possible if GP2/T0CKI isn't being used; it is not a general solution.

The data sheet notes that “the user can work around this by writing an adjusted value to the TMR0 register.” In other words, if you wanted to count 100 cycles, you should preload the value 158, not 156, because the timer does not increment for two cycles after the new value is written.

- Preloading a value in this way only gives accurate results when the prescaler is not used.

The prescaler is a counter, which is not directly accessible. Whenever a value is written to TMR0, the prescaler is cleared. It will then not be incremented for the next two instruction cycles.

For example, if the prescale ratio is set to 1:8, Timer0 normally increments every eight instruction clocks. So when a value is written to TMR0, ten instruction clocks (two plus the normal eight) will elapse before TMR0 is incremented.

This means that, if you were using a 1:8 prescaler, and you preloaded a value to 156 to TMR0, the timer will overflow after 802 instruction cycles, not the 800 cycles (8×100) that you probably intended. Increasing the preloaded value to compensate for these extra two instruction cycles doesn't help – a value of 157 will cause an overflow after 794 cycles ($8 \times 99 + 2$), not 800.

To accurately compensate for the timer being inhibited after TMR0 is written, the prescaler should not be used.

- Some time will have elapsed between the timer overflow and the instruction where you load the new value into the timer.

This is especially true when the timer is being updated within an interrupt service routine; there is some latency between the timer overflow event and the ISR being called (two instruction cycles on a mid-range PIC), and then the ISR must save the processor context, and potentially determine the source of the interrupt, before loading a new value into the timer.

It's possible to account for this latency, by adjusting the value to be loaded into the timer, but only if there is no other interrupt source which may delay the timer interrupt from being triggered. If another interrupt is being serviced when the timer overflows, some unknown amount of time will elapse before the timer interrupt begins – it would be very difficult to allow for that.

Luckily, it is not difficult to avoid all these problems!

To use Timer0 to provide a precise time-base to drive an interrupt:

- Do not use the prescaler (assign it to the watchdog timer).
- Do not load a fixed start value into the timer.

Instead, add an offset to the current timer value, making the timer “skip forward” by an appropriate amount, shortening the timer cycle from 256 counts to whatever period you require.

- Adjust the offset to allow for the fact that the timer is inhibited for two cycles after it is written, and that the timer increments once (if no prescaler is used) during the add instruction.

This means that the offset to be added must be 3 cycles larger than you may expect, to achieve a given timer period.

For example, to make Timer0 overflow after 250 cycles, instead of the usual 256 cycles, with no prescaler, you would use:

```
movlw    .256-.250+.3    ; add value to Timer0
banksel TMR0            ; for overflow after 250 counts
addwf    TMR0, f
```

This needs to be done after every Timer0 overflow (i.e. within the interrupt service routine), so that the interrupt is triggered precisely every 250 instruction cycles.

Recall that, with a 4 MHz processor clock, TMR0 will increment every 1 μ s.

If we adjust TMR0 in the ISR as shown above, the interrupt will be triggered every 250 μ s.

Toggling the LED every 500 ms means toggling after every $500 \text{ ms} \div 250 \mu\text{s} = 2000$ interrupts.

This means that the ISR must be able to count to 2000, so that it can toggle the LED after 2000 interrupts. And since a single 8-bit variable can only hold a count up to 255, we need more than a single 8-bit variable, so that we can count up to 2000.

This could be done by using two registers to implement a single 16-bit variable (see [baseline lesson 11](#)).

However, a more useful approach in this case is to recognise that, if we count 40 interrupts, exactly 10 ms will have elapsed, since $10 \text{ ms} = 40 \times 250 \mu\text{s}$. This makes it easy to schedule an operation (such as polling inputs, as we'll see later) every 10 ms – often a convenient time base.

We can then use a second variable to count 10 ms periods. After every $50 \times 10 \text{ ms}$, 500 ms has elapsed, and the LED should be toggled.

So to count in units of 10 ms, we need two variables:

```
cnt_t0      res 1          ; counts timer0 interrupts
                ; (decremented by ISR every 250 us)
cnt_10ms    res 1          ; counts 10 ms periods
                ; (decremented by ISR every 10 ms)
```

We use the first to count for 40 interrupts, to generate a 10 ms time base:

```
decfsz cnt_t0, f          ; decrement interrupt count
goto isr_end              ; when count = 0 (every 40 interrupts = 10 ms)
movlw .40                  ; reload count
movwf cnt_t0
```

Note again that it's often easiest to use `decfsz` to count a fixed number of iterations (40, in this case).

Then we can count for 50 of these 10 ms periods, in the same way:

```
decfsz cnt_10ms, f        ; decrement 10 ms period count
goto isr_end              ; when count = 0 (every 50 times = 500 ms)
movlw .50                  ; reload count
movwf cnt_10ms
```

After $50 \times 10 \text{ ms} = 500 \text{ ms}$, we can toggle the LED, as we did before:

```
movf sGPIO, w             ; toggle LED
xorlw 1<<nLED              ; using shadow register
movwf sGPIO
```

Of course, in the initialisation part of the main program, we need to configure Timer0 with no prescaler:

```
movlw b'11001000'        ; configure Timer0:
                ; --0----- timer mode (T0CS = 0)
                ; ----1---- no prescaling (PSA = 1)
                ; (prescaler assigned to WDT)
banksel OPTION_REG       ; -> increment TMR0 every 1 us
movwf OPTION_REG
```

And we should initialise the variables used above:

```
movlw .40                ; timer0 overflow count = 40
movwf cnt_t0
movlw .50                ; 10 ms period count = 50
movwf cnt_10ms
```

With these modifications in place, the LED will now flash with a frequency of exactly 1 Hz, assuming that the processor clock is exactly 4 MHz (which, since we are using the internal RC oscillator, will not be the case; it's not that accurate. Nevertheless, the LED flashes every 4,000,000 processor cycles, precisely).

Complete program

Here is how the code fragments above fit together:

```

;*****
;
; Description: Lesson 6 example 1c
;
; Demonstrates use of Timer0 interrupt to perform a background task
;
; Flash an LED at exactly 1 Hz (50% duty cycle).
;
;*****
;
; Pin assignments:
; GP2 = flashing LED
;
;*****

list p=12F629
#include <p12F629.inc>

errorlevel -302 ; no "register not in bank 0" warnings
errorlevel -312 ; no "page or bank selection not needed" messages

;***** CONFIGURATION
; ext reset, no code or data protect, no brownout detect,
; no watchdog, power-up timer, 4 Mhz int clock
__CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

; pin assignments
constant nLED=2 ; flashing LED on GP2

;***** VARIABLE DEFINITIONS
CONTEXT UDATA_SHR ; variables used for context saving
cs_W res 1
cs_STATUS res 1

GENVAR UDATA_SHR ; general variables
sGPIO res 1 ; shadow copy of GPIO
cnt_t0 res 1 ; counts timer0 interrupts
; (decremented by ISR every 250 us)
cnt_10ms res 1 ; counts 10 ms periods
; (decremented by ISR every 10 ms)

;***** RESET VECTOR *****
RESET CODE 0x0000 ; processor reset vector
pagesel start
goto start

;***** INTERRUPT SERVICE ROUTINE *****

```

```

ISR      CODE      0x0004
; *** Save context
movwf   cs_W           ; save W
movf    STATUS,w       ; save STATUS
movwf   cs_STATUS

; *** Service Timer0 interrupt
;
;   TMR0 overflows every 250 us
;
;   Flashes LED at 1 Hz by toggling on every 2000th interrupt
;   (every 500 ms)
;
;   (only Timer0 interrupts are enabled)
;
movlw   .256-.250+.3   ; add value to Timer0
banksel TMR0           ; for overflow after 250 counts
addwf   TMR0,f
bcf     INTCON,T0IF    ; clear interrupt flag

; count for 10 ms (40 interrupts x 250 us)
decfsz  cnt_t0,f       ; decrement interrupt count
goto    isr_end        ; when count = 0 (every 40 interrupts = 10 ms)
movlw   .40            ; reload count
movwf   cnt_t0

; toggle LED every 500 ms
decfsz  cnt_10ms,f     ; decrement 10 ms period count
goto    isr_end        ; when count = 0 (every 50 times = 500 ms)
movlw   .50            ; reload count
movwf   cnt_10ms

movf    sGPIO,w        ; toggle LED
xorlw   1<<nLED        ; using shadow register
movwf   sGPIO

isr_end ; *** Restore context then return
movf    cs_STATUS,w    ; restore STATUS
movwf   STATUS
swapf   cs_W,f         ; restore W
swapf   cs_W,w
retfie

;***** MAIN PROGRAM *****
MAIN    CODE
start   ; calibrate internal RC oscillator
        call    0x03FF ; retrieve factory calibration value
        banksel OSCCAL ; (stored at 0x3FF as a retlw k)
        movwf   OSCCAL ; then update OSCCAL

;***** Initialisation

; configure port
banksel GPIO
clrf    GPIO           ; start with all LEDs off
clrf    sGPIO          ; update shadow
movlw   ~(1<<nLED)     ; configure LED pin (only) as an output
banksel TRISIO
movwf   TRISIO

```



```

; configure timer
movlw    b'11001000'    ; configure Timer0:
                    ; --0-----    timer mode (TOCS = 0)
                    ; ----1----    no prescaling (PSA = 1)
                    ;                (prescaler assigned to WDT)
banksel  OPTION_REG    ; -> increment TMR0 every 1 us
movwf   OPTION_REG

; initialise variables
movlw   .40            ; timer0 overflow count = 40
movwf  cnt_t0
movlw   .50            ; 10 ms period count = 50
movwf  cnt_10ms

; enable interrupts
movlw   1<<GIE|1<<T0IE ; enable Timer0 and global interrupts
movwf  INTCON

;***** Main loop
main_loop
    ; continually copy shadow GPIO to port
    movf    sGPIO,w
    banksel GPIO
    movwf   GPIO

    ; repeat forever
    goto   main_loop

END

```

Example 2: Flash LED while responding to input

Now that we have a timer-driven interrupt flashing the LED on GP2 at 1 Hz, that flashing will continue independently, “on its own”, regardless of whatever the main program code is doing.⁵

This is the main reason for using a timer interrupt to drive a background process like this; once the process is set up, you do not need to worry about maintaining it in the main code. It may seem complex to set up the interrupt code, but, once done, it makes your main code much easier to write.

To illustrate this, we can re-implement example 2 from [lesson 4](#), where we the LED on GP1 is lit whenever the pushbutton is pressed, while the LED on GP2 continues to flash steadily at 1 Hz.

In [lesson 4](#), we used this simple piece of code to read the pushbutton and light the LED on GP1 only when it is pressed:

```

banksel  GPIO          ;      check and respond to button press
bcf     sGPIO,GP1     ;      assume button up -> indicator LED off
btfss   GPIO,GP3     ;      if button pressed (GP3 low)
bsf     sGPIO,GP1     ;      turn on indicator LED

movf    sGPIO,w       ;      update port (copy shadow to GPIO)
movwf   GPIO

```

⁵ Assuming of course that the main program continues to regularly copy the shadow register to GPIO, and does not disable the Timer0 interrupt, nor change the configuration of Timer0.

In the main loop in example 1, above, we are doing nothing but copy the shadow register to GPIO:

```
main_loop
    ; continually copy shadow GPIO to port
    movf    sGPIO,w
    banksel GPIO
    movwf   GPIO

    ; repeat forever
    goto   main_loop
```

All we need do, then, is to insert the pushbutton-handling code into the main loop:

```
main_loop
    ; check and respond to button press
    banksel GPIO
    bcf     sGPIO,nB_LED    ; assume button up -> LED off
    btfss  GPIO,nBUTTON    ; if button pressed (low)
    bsf     sGPIO,nB_LED    ; turn on indicator LED

    ; continually copy shadow GPIO to port
    movf    sGPIO,w
    movwf   GPIO

    ; repeat forever
    goto   main_loop
```

And of course you could add any other code to the main loop, in the same way. There is no need to be “aware” of the interrupt-driven process; it runs quite independently.

Note that symbols have been used here, which were defined as:

```
constant    nB_LED=1          ; "button pressed" indicator LED on GP1
constant    nF_LED=2          ; flashing LED on GP2
constant    nBUTTON=3         ; pushbutton on GP3
```

The only other change that has to be made to the code in example 1 is to configure both of the LED pins as outputs:

```
movlw      ~(1<<nB_LED|1<<nF_LED)    ; configure LED pins as outputs
banksel    TRISIO
movwf     TRISIO
```

No changes are needed within the interrupt service routine.

Complete program

Although the changes to the code in example 1 are minor, here is how they fit together:

```
*****
;
; Description:      Lesson 6 example 2
;
; Demonstrates use of Timer0 interrupt to perform a background task
; while performing other actions in response to changing inputs
;
; One LED simply flashes at 1 Hz (50% duty cycle).
; The other LED is only lit when the pushbutton is pressed.
*****
```

```

;*****
; Pin assignments:
; GP1 = "button pressed" indicator LED
; GP2 = flashing LED
; GP3 = pushbutton switch (active low)
;
;*****

list      p=12F629
#include   <p12F629.inc>

errorlevel -302    ; no "register not in bank 0" warnings
errorlevel -312    ; no "page or bank selection not needed" messages

;***** CONFIGURATION
                ; int reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4 Mhz int clock
__CONFIG      _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

; pin assignments
constant      nB_LED=1          ; "button pressed" indicator LED on GP1
constant      nF_LED=2          ; flashing LED on GP2
constant      nBUTTON=3         ; pushbutton on GP3

;***** VARIABLE DEFINITIONS
CONTEXT       UDATA_SHR        ; variables used for context saving
cs_W          res 1
cs_STATUS     res 1

GENVAR        UDATA_SHR        ; general variables
sGPIO         res 1            ; shadow copy of GPIO
cnt_t0        res 1            ; counts timer0 interrupts
                                ; (decremented by ISR every 250 us)
cnt_10ms      res 1            ; counts 10 ms periods
                                ; (decremented by ISR every 10 ms)

;***** RESET VECTOR *****
RESET         CODE    0x0000    ; processor reset vector
              pagesel start
              goto     start

;***** INTERRUPT SERVICE ROUTINE *****
ISR           CODE    0x0004
; *** Save context
movwf        cs_W          ; save W
movf         STATUS,w      ; save STATUS
movwf        cs_STATUS

; *** Service Timer0 interrupt
;
; TMR0 overflows every 250 us
;
; Flashes LED at 1 Hz by toggling on every 2000th interrupt
; (every 500 ms)
;
; (only Timer0 interrupts are enabled)

```

```

;
movlw    .256-.250+.3    ; add value to Timer0
banksel  TMR0           ; for overflow after 250 counts
addwf    TMR0,f
bcf      INTCON,T0IF    ; clear interrupt flag

; count for 10 ms (40 interrupts x 250 us)
decfsz   cnt_t0,f      ; decrement interrupt count
goto     isr_end       ; when count = 0 (every 40 interrupts = 10 ms)
movlw    .40           ; reload count
movwf    cnt_t0

; toggle LED every 500 ms
decfsz   cnt_10ms,f   ; decrement 10 ms period count
goto     isr_end       ; when count = 0 (every 50 times = 500 ms)
movlw    .50           ; reload count
movwf    cnt_10ms

movf     sGPIO,w       ; toggle LED
xorlw    1<<nF_LED     ; using shadow register
movwf    sGPIO

isr_end ; *** Restore context then return
movf     cs_STATUS,w   ; restore STATUS
movwf    STATUS
swapf    cs_W,f        ; restore W
swapf    cs_W,w
retfie

;***** MAIN PROGRAM *****
MAIN     CODE
start    ; calibrate internal RC oscillator
call     0x03FF        ; retrieve factory calibration value
banksel  OSCCAL        ; (stored at 0x3FF as a retlw k)
movwf    OSCCAL        ; then update OSCCAL

;***** Initialisation

; configure port
banksel  GPIO
clrf     GPIO          ; start with all LEDs off
clrf     sGPIO         ; update shadow
movlw    ~(1<<nB_LED|1<<nF_LED) ; configure LED pins as outputs
banksel  TRISIO
movwf    TRISIO

; configure timer
movlw    b'11001000'   ; configure Timer0:
; --0-----          timer mode (T0CS = 0)
; ----1----          no prescaling (PSA = 1)
; (prescaler assigned to WDT)
banksel  OPTION_REG    ; -> increment TMR0 every 1 us
movwf    OPTION_REG

; initialise variables
movlw    .40           ; timer0 overflow count = 40
movwf    cnt_t0
movlw    .50           ; 10 ms period count = 50
movwf    cnt_10ms

```

```

; enable interrupts
movlw 1<<GIE|1<<T0IE ; enable Timer0 and global interrupts
movwf INTCON

;***** Main loop
main_loop
; check and respond to button press
banksel GPIO
bcf sGPIO,nB_LED ; assume button up -> LED off
btfss GPIO,nBUTTON ; if button pressed (low)
bsf sGPIO,nB_LED ; turn on indicator LED

; continually copy shadow GPIO to port
movf sGPIO,w
movwf GPIO

; repeat forever
goto main_loop

END

```

Example 3: Switch debouncing

[Lesson 3](#) explored the topic of switch bounce, and described a counting algorithm to address it, which was expressed as:

```

count = 0
while count < max_samples
  delay sample_time
  if input = required_state
    count = count + 1
  else
    count = 0
end

```

The change in switch state is only accepted when the new state has been continually seen for at least some minimum period, for example 20 ms. This debounce period is measured by incrementing a count while sampling the state of the switch, at a steady rate, such as every 1 ms.

“Continually ... sampling ... at a steady rate” sounds like the type of task that could be performed by a regular timer interrupt, and indeed it is common to use interrupts to continually sample and debounce inputs.

Although a number of debouncing algorithms exist, offering varying levels of sophistication, the counting algorithm presented above is effective and is easy to implement in an interrupt service routine.

But when this algorithm was implemented before, in lessons 3 and 4, separate routines were used to wait for and debounce “button up” and “button down” (low → high and high → low transitions). That approach isn’t appropriate in an ISR, since it has to run independently of the main program; it can’t know what type of transition the main program is waiting for. If we want to detect and debounce both types of transitions, the ISR needs to look for any change in state, and debounce it. And then it needs to have some way of reporting the fact that an input transition (change in switch state) has occurred, in case the main program chooses to act on it.

You could have the ISR respond to and act upon switch changes, but this isn’t normally done unless the event has to be responded to very quickly; it is generally best to keep the interrupt handling code short, so that the ISR finishes quickly, in case another, perhaps more important, interrupt is pending.

Normally, this type of signalling from the ISR to the main program is done via a flag which is set by the ISR to indicate that an event has occurred. The main program then polls this flag and responds to the event when it is ready to do so.

In this case, we would need a ‘switch state has changed’ flag.

We also need a flag, or variable, to hold the “debounced”, or most recently accepted state of the switch input. The ISR can then periodically compare the current “raw” switch input with the saved “debounced” input, to determine whether the switch state has changed.

To demonstrate this approach, we’ll re-implement example 2 from [lesson 3](#), where the LED on GP1 is toggled each time the pushbutton on GP3 is pressed.

We can re-use and modify the framework from the examples above, where we flashed an LED at 1 Hz.

In those examples, the ISR was triggered every 250 μ s, which in turn counted interrupts, to create a 10 ms time base.

However, for sampling a switch input, 250 μ s is a little too short (the more often you sample an input, the more time overall is spent in the ISR, leaving less time for the main program), while 10 ms is too long. Many switches stop bouncing within 20 ms, so if you sample every 10 ms, and have a debounce period of 20 ms, you’ll be basing the decision that the switch is stable on only two samples – and two samples in a row might be a “fluke”; it’s not enough to be sure that the bouncing (or glitches due to EMI) has finished.

Typically a sample rate between 1 ms and 5 ms is recommended; we’ll use 2 ms here.

So the timing section of the ISR becomes:

```

; *** Service Timer0 interrupt
;
;   TMR0 overflows every 250 clocks = 250 us
;
movlw  .256-.250+.3    ; add value to Timer0
banksel TMR0          ;   for overflow after 250 counts
addwf  TMR0,f
bcf    INTCON,T0IF    ; clear interrupt flag

; count interrupts to generate 2 ms tick
decfsz cnt_t0,f      ; decrement interrupt count
goto   isr_end       ; when count = 0
movlw  .2000/.250     ;   reload count for next 2 ms period
movwf  cnt_t0         ;   (2ms / 250 us/interrupt)

```

We also need some variables, discussed above, for the debounce algorithm:

```

PB_dbstate  res 1          ; bit 3 = debounced pushbutton state
                ;   (0 = pressed, 1 = released)
PB_change   res 1          ; bit 3 = flag indicating pushbutton state change
                ;   (1 = new debounced state)
cnt_db      res 1          ; debounce counter

```

Note that, because only a single bit is needed to represent the switch state, or to flag that the switch has changed, we can choose to use any of the bits within these variables.

It’s most convenient (the coding is simplified) if we use bit 3 of the `PB_dbstate` variable to represent the debounced state of the switch on GP3. This implies that this technique could be extended to debounce other switches at the same time, although in practice, another technique, based on “vertical counters” is more commonly used when debouncing multiple switches. We’ll look at it in a later lesson.

Of course these variables should be initialised in the main program:

```
movlw    1<<nBUTTON      ; initial pushbutton state = released
movwf    PB_dbstate
clrf     cnt_db          ; debounce counter = 0
clrf     PB_change       ; pushbutton change flag = 0
```

It is a good idea to define the debounce period as a constant, to make it easier to adapt the code for switches with different characteristics:

```
constant    MAX_DB_CNT=.20/.2 ; maximum debounce count =
                                ;   debounce period / sample rate
                                ;   (20 ms debounce period / 2 ms per sample)
```

(of course it would be cleaner still to define the debounce period and sample rate as constants, and to derive the maximum debounce count and sample timing from them – but in a short program like this it's not difficult to see how these things relate to each other, especially if it is documented in comments, as above)

Now for the debounce routine, run every 2 ms as part of the interrupt service routine.

First, we need to determine whether the pushbutton has changed (pressed or released) since it was last debounced.

To do so, we need to compare GPIO<3> with PB_dbstate<3>, and this means some logic operations:

```
; has raw state changed?
banksel  GPIO
movlw    1<<nBUTTON      ; load raw button state (only) to W
andwf    GPIO,w
xorwf    PB_dbstate,w   ; XOR with last debounced state
btfss    STATUS,Z       ; (result of XOR is zero if same,
goto     state_change   ; so Z flag is clear if state has changed)
```

Note the use of the 'andwf' instruction – “**and W** with file register”, which ANDs the contents of W with the specified register and writes the results to either the register or W.

It is used here to apply a *mask* to GPIO, so that only GPIO<3> (the only bit we are interested in) is transferred to W. Using AND to mask bits in this way was explained in [baseline lesson 8](#).

As we've seen before, XOR can be used to test for equality. Note however that, if we were using any other bits in PB_dbstate, we'd have to mask them out before doing the comparison.

Having determined whether the pushbutton's raw state has changed, we need to deal with both possibilities.

If the pushbutton is still in the last debounced state, all we need to do is reset the debounce counter:

```
; raw pushbutton state has not changed
clrf     cnt_db         ; reset debounce count
goto     debounce_end   ; and exit
```

Otherwise, the pushbutton's state has changed. We need to see whether the change is stable, by counting the number of successive times we've seen it in this new state:

```
state_change
; raw pushbutton state has changed
incf     cnt_db,f       ; increment count
```

And then we need to check whether the maximum count has been reached, to determine whether the switch really has changed state (and has finished bouncing):

```
movlw    MAX_DB_CNT           ; has max count been reached yet?
xorwf    cnt_db,w
btfss    STATUS,Z            ; if not,
goto     debounce_end        ; exit
```

If so, we have a new debounced state, so we can update the variables and flags to reflect this:

```
; accept state as changed
movlw    1<<nBUTTON          ; toggle debounced state
xorwf    PB_dbstate,f
clrf     cnt_db              ; reset debounce count
bsf      PB_change,nBUTTON   ; set pushbutton changed flag
```

The main program can then poll this PB_change flag, to see whether the button has changed state:

```
; check for debounced button press
btfss    PB_change,nBUTTON   ; has button state changed?
goto     pb_press_end
```

If the button has changed state, we need to refer to the PB_dbstate variable, to see whether it the new state is “up” or “down” (pressed); we only want to toggle the LED when the button is pressed, not when it is released:

```
btfsc    PB_dbstate,nBUTTON  ; is button pressed (low)?
goto     pb_press_end
```

When we know that the button has been pressed, we can toggle the LED, using the shadow copy of GPIO, as we’ve done before:

```
; handle button press
movlw    1<<nB_LED           ; toggle indicator LED
xorwf    sGPIO,f            ; using shadow register
```

And finally, now that we’ve detected and responded to the button press, we need to clear the state change flag, to be ready for the next change:

```
bcf      PB_change,nBUTTON   ; clear button change flag
```

And that’s all.

It’s relatively complex, compared with the equivalent code we saw in lessons [3](#) and [4](#), but most of that complexity is “hidden” in the ISR; the code in the main program loop is quite simple, making it easier to do more within the main program, without having to poll and debounce switches – something that the ISR can take care of in the background. This interrupt-based approach also has the advantage that switch changes are detected quickly, while the main program does not have to respond to them immediately.

Complete program

Here is the complete “toggle an LED on pushbutton press” program:

```
;*****
;
; Description:      Lesson 6 example 3
;
; Demonstrates use of Timer0 interrupt to implement
; counting debounce algorithm
;*****
```



```

; Toggles LED when pushbutton is pressed (high -> low transition)      *
;                                                                           *
;*****
; Pin assignments:                                                         *
;     GP1 = indicator LED                                                 *
;     GP3 = pushbutton (active low)                                       *
;*****

list      p=12F629
#include   <p12F629.inc>

errorlevel -302      ; no "register not in bank 0" warnings
errorlevel -312      ; no "page or bank selection not needed" messages

;***** CONFIGURATION
                ; int reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4 Mhz int clock
__CONFIG      _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

; pin assignments
constant      nB_LED=1                ; "button pressed" indicator LED on GP1
constant      nBUTTON=3                ; pushbutton on GP3

;***** CONSTANTS
constant      MAX_DB_CNT=.20/.2        ; max debounce count
                                                ; = debounce period / sample rate
                                                ; (20 ms / 2 ms per sample)

;***** VARIABLE DEFINITIONS
CONTEXT       UDATA_SHR                ; variables used for context saving
cs_W          res 1
cs_STATUS     res 1

GENVAR        UDATA_SHR                ; general variables
sGPIO         res 1                    ; shadow copy of GPIO
cnt_t0        res 1                    ; counts timer0 interrupts
                                                ; (decremented by ISR every 250 us)
PB_dbstate    res 1                    ; bit 3 = debounced pushbutton state
                                                ; (0 = pressed, 1 = released)
PB_change     res 1                    ; bit 3 = flag indicating pushbutton state
change
                                                ; (1 = new debounced state)
cnt_db        res 1                    ; debounce counter

;***** RESET VECTOR *****
RESET        CODE    0x0000            ; processor reset vector
            pagesel start
            goto     start

;***** INTERRUPT SERVICE ROUTINE *****
ISR          CODE    0x0004
            ; *** Save context
            movwf   cs_W                ; save W
            movf    STATUS,w           ; save STATUS
            movwf   cs_STATUS

```

```

; *** Service Timer0 interrupt
;
;   TMR0 overflows every 250 us
;
;   Debounces pushbutton:
;     samples every 2 ms (every 8th interrupt)
;     -> PB_dbstate<3> = debounced state
;         PB_change<3> = change flag (1 = new debounced state)
;
;   (only Timer0 interrupts are enabled)
;
movlw  .256-.250+.3    ; add value to Timer0
banksel TMR0          ; for overflow after 250 counts
addwf  TMR0,f
bcf    INTCON,T0IF    ; clear interrupt flag

; count interrupts to generate 2 ms tick
decfsz cnt_t0,f      ; decrement interrupt count
goto   isr_end       ; when count = 0
movlw  .2000/.250    ; reload count for next 2 ms period
movwf  cnt_t0        ; (2ms / 250us/interrupt)

; Debounce pushbutton (every 2 ms)
; use counting algorithm: accept change in state
; only if new state is seen a number of times in succession

; has raw state changed?
banksel GPIO
movlw  1<<nBUTTON    ; load raw button state (only) to W
andwf  GPIO,w
xorwf  PB_dbstate,w  ; XOR with last debounced state
btfss  STATUS,Z      ; (result of XOR is zero if same,
goto   state_change  ; so Z flag is clear if state has changed)

; raw pushbutton state has not changed
clrf   cnt_db        ; reset debounce count
goto   debounce_end  ; and exit

state_change
; raw pushbutton state has changed
incf   cnt_db,f      ; increment count
movlw  MAX_DB_CNT    ; has max count been reached yet?
xorwf  cnt_db,w
btfss  STATUS,Z      ; if not,
goto   debounce_end  ; exit

; accept new state as changed
movlw  1<<nBUTTON    ; toggle debounced state
xorwf  PB_dbstate,f
clrf   cnt_db        ; reset debounce count
bsf    PB_change,nBUTTON ; set pushbutton changed flag
; (polled and cleared in main loop)

debounce_end

isr_end ; *** Restore context then return
movf   cs_STATUS,w   ; restore STATUS
movwf  STATUS
swapf  cs_W,f        ; restore W
swapf  cs_W,w
retfie

```

```

;***** MAIN PROGRAM *****
MAIN    CODE
start   ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ; (stored at 0x3FF as a retlw k)
        movwf   OSCCAL          ; then update OSCCAL

;***** Initialisation

        ; configure port
        banksel GPIO
        clrf    GPIO            ; start with all LEDs off
        clrf    sGPIO           ; update shadow
        movlw   ~(1<<nB_LED)     ; configure LED pin as output
        banksel TRISIO
        movwf   TRISIO

        ; configure timer
        movlw   b'11001000'      ; configure Timer0:
                ; --0-----      timer mode (T0CS = 0)
                ; ----1----      no prescaling (PSA = 1)
                                ; (prescaler assigned to WDT)
        banksel OPTION_REG      ; -> increment TMR0 every 1 us
        movwf   OPTION_REG

        ; initialise variables
        movlw   .2000/.250       ; timer0 overflow count = 2ms / 250us/overflow
        movwf   cnt_t0           ; (-> 2 ms per switch sample)
        movlw   1<<nBUTTON        ; initial pushbutton state = released
        movwf   PB_dbstate
        clrf    cnt_db           ; debounce counter = 0
        clrf    PB_change       ; pushbutton change flag = 0

        ; enable interrupts
        movlw   1<<GIE|1<<T0IE   ; enable Timer0 and global interrupts
        movwf   INTCON

;***** Main loop
main_loop
        ; check for debounced button press
        btfss  PB_change,nBUTTON ; has button state changed?
        goto   pb_press_end
        btfsc  PB_dbstate,nBUTTON ; is button pressed (low)?
        goto   pb_press_end

        ; handle button press
        movlw   1<<nB_LED         ; toggle indicator LED
        xorwf   sGPIO,f          ; using shadow register
        bcf    PB_change,nBUTTON ; clear button change flag
pb_press_end

        ; continually copy shadow GPIO to port
        banksel GPIO
        movf    sGPIO,w
        movwf   GPIO

        ; repeat forever
        goto   main_loop

END

```

Example 4: Switch debouncing while flashing an LED

Given that the previous example on switch debouncing was built on the framework of the earlier LED flashing examples, it's not difficult to add the LED flashing code back into the interrupt service routine, demonstrating how a single timer-driven interrupt can be used to schedule multiple concurrent tasks.

Firstly, as before, we need a counter, so that we can count up to 500 ms:

```
cnt_2ms    res 1                ; counts 2 ms periods
                                ; (decremented by ISR every 2 ms)
```

Note that this counter is intended to count periods of 2 ms each; this is the same as the switch sample period from the previous example. That's not a coincidence! It makes sense to make use of common time bases if possible, to avoid adding unnecessary code. And this is why a sample period of 2 ms was chosen in the last example, instead of 1 ms – to generate a 500 ms delay by counting 1 ms periods, we'd need to count to 500, and that's not possible with a single 8-bit variable. By using a time base of 2 ms, we not only have an appropriate period for sampling the switch, but we only need a single 8-bit counter to generate a 500 ms delay, since $2\text{ ms} \times 250 = 500\text{ ms}$, and we can count to 250 with an 8-bit variable.

We should of course initialise this counter, in the main program, before it is used:

```
movlw     .500/.2                ; 2 ms period count = 500ms / 2ms
movwf    cnt_2ms                 ; (-> toggle LED every 500 ms)
```

Then, either before or after the debounce routine in the ISR (it doesn't matter, since they both need to run every 2 ms), we need some code to count 2 ms periods, to create a 500 ms delay:

```
; toggle LED every 500 ms
decfsz   cnt_2ms,f              ; decrement 2 ms period count
goto     toggle_end            ; when count = 0
movlw    .500/.2                ; reload count for next 500 ms period
movwf    cnt_2ms               ; (500ms / 2ms/tick)
```

And finally, when 500 ms has elapsed, we toggle the LED, using the shadow copy of GPIO, as before:

```
movf     sGPIO,w                ; toggle LED
xorlw    1<<nF_LED              ; using shadow register
movwf    sGPIO
```

Complete interrupt service routine

Most of the code is the same as the previous example, except for the counter variable definition and initialisation, shown above. But here is the new interrupt service routine, so that you can see how the LED toggling code fits in after the debounce routine:

```
***** INTERRUPT SERVICE ROUTINE *****
ISR    CODE    0x0004
        ; *** Save context
        movwf   cs_W                ; save W
        movf    STATUS,w            ; save STATUS
        movwf   cs_STATUS

        ; *** Service Timer0 interrupt
        ;
        ; TMR0 overflows every 250 clocks = 250 us
        ;
        ; Debounces pushbutton:
        ; samples every 2 ms (every 8th interrupt)
        ; -> PB_dbstate<3> = debounced state
        ; PB_change<3> = change flag (1 = new debounced state)
```

```

;
;   Flashes LED at 1 Hz by toggling every 500 ms
;       (every 250th 2 ms period)
;
;   (only Timer0 interrupts are enabled)
;
movlw    .256-.250+.3    ; add value to Timer0
banksel  TMR0            ; for overflow after 250 counts
addwf    TMR0,f
bcf      INTCON,T0IF     ; clear interrupt flag

; count interrupts to generate 2 ms tick
decfsz   cnt_t0,f       ; decrement interrupt count
goto     isr_end        ; when count = 0
movlw    .2000/.250     ; reload count for next 2 ms period
movwf    cnt_t0         ; (2ms / 250us/interrupt)

; Debounce pushbutton (every 2 ms)
; use counting algorithm: accept change in state
; only if new state is seen a number of times in succession

; has raw state changed?
banksel  GPIO
movlw    1<<nBUTTON     ; load raw button state (only) to W
andwf    GPIO,w
xorwf    PB_dbstate,w   ; XOR with last debounced state
btfss    STATUS,Z       ; (result of XOR is zero if same,
goto     state_change   ; so Z flag is clear if state has changed)

; raw pushbutton state has not changed
clrf     cnt_db         ; reset debounce count
goto     debounce_end   ; and exit

state_change
; raw pushbutton state has changed
incf     cnt_db,f       ; increment count
movlw    MAX_DB_CNT     ; has max count been reached yet?
xorwf    cnt_db,w
btfss    STATUS,Z       ; if not,
goto     debounce_end   ; exit

; accept new state as changed
movlw    1<<nBUTTON     ; toggle debounced state
xorwf    PB_dbstate,f
clrf     cnt_db         ; reset debounce count
bsf      PB_change,nBUTTON ; set pushbutton changed flag
; (polled and cleared in main loop)

debounce_end

; toggle LED every 500 ms
decfsz   cnt_2ms,f     ; decrement 2 ms period count
goto     toggle_end    ; when count = 0
movlw    .500/.2       ; reload count for next 500 ms period
movwf    cnt_2ms       ; (500ms / 2ms/tick)

movf     sGPIO,w        ; toggle LED
xorlw    1<<nF_LED      ; using shadow register
movwf    sGPIO

toggle_end

```

```

isr_end ; *** Restore context then return
        movf    cs_STATUS,w      ; restore STATUS
        movwf   STATUS
        swapf   cs_W,f          ; restore W
        swapf   cs_W,w
        retfie

```

External Interrupts

Although polling input pins for changes is effective in many cases, especially in user interfaces, where the human user won't notice a delay of a few milliseconds before a button press is responded to, some situations require a more immediate response.

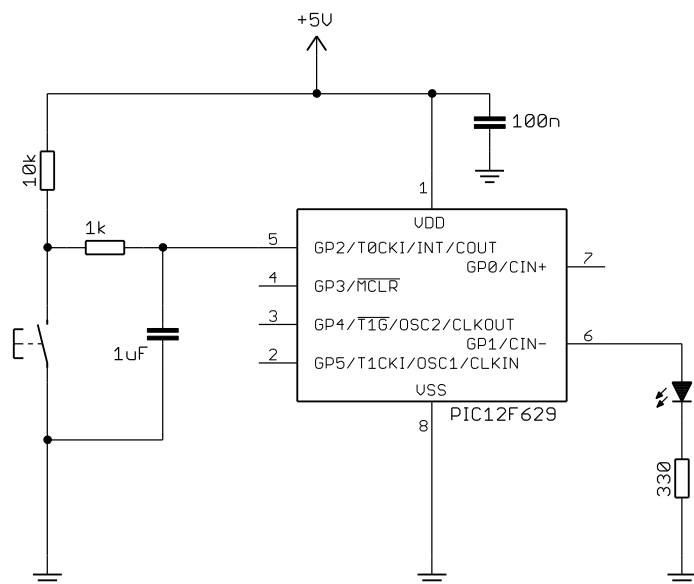
For a very fast response to a digital signal, the external interrupt pin, INT (which shares its pin with GP2) can be used. This pin is *edge-triggered*, meaning that an interrupt will be triggered (if enabled) by a rising or falling transition of the input signal.

Example 5: Using a pushbutton to trigger an external interrupt

To demonstrate how to use external interrupts, we can use a pushbutton to drive the external interrupt pin, and toggle an LED whenever the external interrupt is triggered (i.e. whenever the pushbutton is pressed).

The circuit for this (with the reset switch and its pull-up resistor omitted for clarity) is shown on the right.

It's quite straightforward, but note the capacitor connected across the switch. This is used, in conjunction with the two resistors, to debounce the pushbutton.

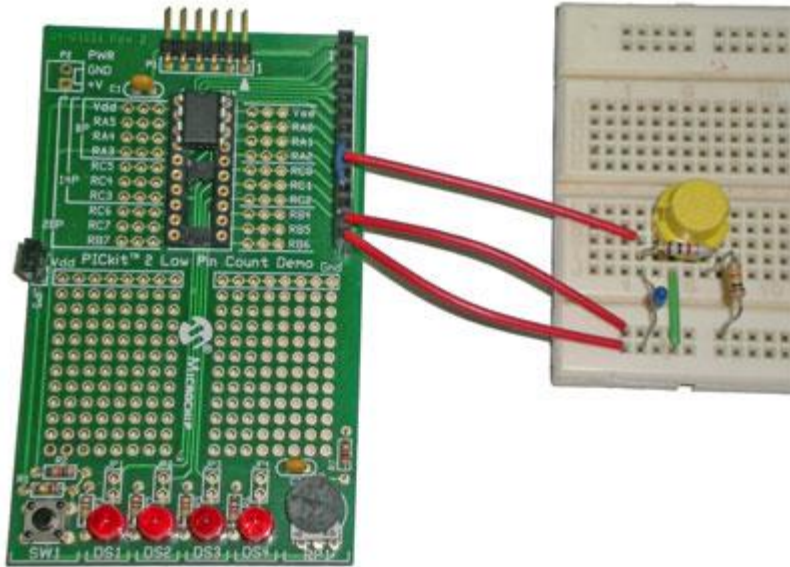


The component values do not have to be exactly as shown. As a guide, the RC time constants should be roughly on the order of the debounce period. The capacitor charges through the 10 kΩ and 1 kΩ resistors, with a time constant of $11 \text{ k}\Omega \times 1 \mu\text{F} = 11 \text{ ms}$.

It discharges, when the button is pressed, with a time constant of $1 \text{ k}\Omega \times 1 \mu\text{F} = 1 \text{ ms}$. These figures are in line with a debounce period of 10 ms or so, but any values similar to these will be ok.

To implement this circuit with the [Gooligum training board](#), close jumpers JP3, JP7 and JP12 to enable the 10 kΩ pull-up resistors on $\overline{\text{MCLR}}$ and GP2 and the LED on GP1.

You also need to add a 1 μF capacitor (supplied with the board) between GP2 and ground. You can do via pins 13 ('GP/RA/RB2') and 16 ('GND') on the 16-pin expansion header. There should be no need to use the solderless breadboard – simply plug the capacitor directly into these header pins.



If you are using Microchip’s Low Pin Count Demo Board, you can build this circuit on a solderless breadboard with a pushbutton switch, resistors and capacitor (which you will have to supply yourself), connected to the 14-pin header on the demo board, as shown on the left.

In this picture, a link has been added, from pin 8 to pin 11 on the header, so that the LED labelled ‘DS2’ on the demo board lights up when GP1 goes high.

When hardware debouncing was discussed in [baseline lesson 4](#), it was pointed out that this type of simple RC filter is only effective when driving a Schmitt trigger input. Luckily, the 12F629’s INT input is a Schmitt trigger type, so this simple form of hardware debouncing is quite adequate.

Of course, the switch debouncing could be done in software, but it is difficult to do for an edge-triggered interrupt, while retaining a fast response (e.g. a short glitch will trigger the interrupt, but should really be ignored – this simple circuit will effectively filter out such glitches).

As mentioned above, the external interrupt can be triggered on either the rising or falling edge of the signal on the INT pin.

The type of edge is selected by the INTEDG bit in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION_REG	GPPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0

INTEDG = 0 selects interrupt on falling edge.

INTEDG = 1 selects interrupt on rising edge.

In this example, since we want the LED to toggle as soon as the pushbutton is pressed (which in this circuit creates a high → low transition on INT), we need to select the falling edge:

```

; configure external interrupt
banksel OPTION_REG
bcf    OPTION_REG,INTEDG    ; trigger on falling edge
    
```

We then need to enable the external interrupt, by setting the INTE bit (as well as GIE, as always):

```

; enable interrupts
movlw  1<<GIE|1<<INTE    ; enable external and global interrupts
movwf  INTCON
    
```

Finally, within the ISR, we need to service the external interrupt.

Since the INT pin is the only interrupt source enabled, it is safe to assume that every interrupt is externally triggered, so all we need to do is clear the INTF interrupt flag (recall that the interrupt flag for any interrupt source has to be cleared, when that interrupt has been serviced), and toggle the LED:

```

    bcf      INTCON,INTF          ; clear interrupt flag

    ; toggle LED
    movlw   1<<nB_LED            ; toggle indicator LED
    xorwf   sGPIO,f              ; using shadow register

```

The shadow register is then copied to GPIO in the main loop, as in the earlier examples.

Complete program

Here is how these code fragments fit together with code from the previous examples:

```

;*****
;
; Description:      Lesson 6 example 5
;
; Demonstrates use of external interrupt (INT pin)
;
; Toggles LED when pushbutton on INT is pressed
; (high -> low transition)
;
;*****
;
; Pin assignments:
;   GP1 = indicator LED
;   INT = pushbutton (active low)
;
;*****

list      p=12F629
#include   <p12F629.inc>

errorlevel -302      ; no "register not in bank 0" warnings
errorlevel -312      ; no "page or bank selection not needed" messages

;***** CONFIGURATION
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
    _PWRTE_ON & _INTRC_OSC_NOCLKOUT

; pin assignments
    constant    nB_LED=1          ; "button pressed" indicator LED on GP1

;***** VARIABLE DEFINITIONS
CONTEXT        UDATA_SHR          ; variables used for context saving
cs_W           res 1
cs_STATUS      res 1

GENVAR         UDATA_SHR          ; general variables
sGPIO          res 1              ; shadow copy of GPIO

;***** RESET VECTOR *****

```



```

RESET   CODE    0x0000           ; processor reset vector
        pagesel start
        goto    start

;***** INTERRUPT SERVICE ROUTINE *****
ISR     CODE    0x0004
        ; *** Save context
        movwf  cs_W              ; save W
        movf   STATUS,w         ; save STATUS
        movwf  cs_STATUS

        ; *** Service external interrupt
        ;
        ;   Triggered on high -> low transition on INT pin
        ;   caused by externally debounced pushbutton press
        ;
        ;   Toggles LED on every high -> low transition
        ;
        ;   (only external interrupts are enabled)
        ;
        bcf    INTCON,INTF      ; clear interrupt flag

        ; toggle LED
        movlw  1<<nB_LED       ; toggle indicator LED
        xorwf  sGPIO,f         ; using shadow register

ISR_end ; *** Restore context then return
        movf   cs_STATUS,w     ; restore STATUS
        movwf  STATUS
        swapf  cs_W,f         ; restore W
        swapf  cs_W,w
        retfie

;***** MAIN PROGRAM *****
MAIN    CODE
start   ; calibrate internal RC oscillator
        call  0x03FF          ; retrieve factory calibration value
        banksel OSCCAL       ; (stored at 0x3FF as a retlw k)
        movwf OSCCAL        ; then update OSCCAL

;***** Initialisation

        ; configure port
        banksel GPIO
        clrf  GPIO           ; start with all LEDs off
        clrf  sGPIO          ; update shadow
        movlw ~(1<<nB_LED)    ; configure LED pin (only) as an output
        banksel TRISIO
        movwf TRISIO

        ; configure external interrupt
        banksel OPTION_REG
        bcf   OPTION_REG,INTEDG ; trigger on falling edge (INTEDG = 0)

        ; enable interrupts
        movlw 1<<GIE|1<<INTE ; enable external and global interrupts
        movwf INTCON

```

```

;***** Main loop
main_loop
    ; continually copy shadow GPIO to port
    movf    sGPIO,w
    banksel GPIO
    movwf  GPIO

    ; repeat forever
    goto   main_loop

END

```

Example 6: Multiple interrupt sources

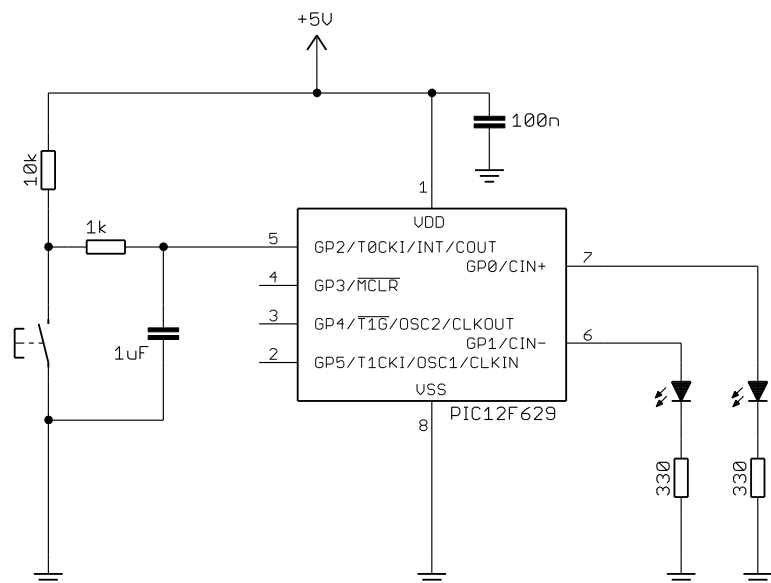
So far we've only used a single interrupt source, but it is common for more than one source to be active; for example, one or more timers scheduling background tasks, while servicing events such as external interrupts.

To demonstrate this, we can combine the two interrupt sources used in this lesson, with a Timer0 interrupt flashing one LED, while the external interrupt is used to toggle another LED.

This means adding an LED to the circuit in the previous example, as shown on the right.

If you have the [Gooligum training board](#), leave it set up as in the last example, but close jumper JP11 to enable the LED on GP0.

We'll flash the LED on GP0 at 1 Hz, and toggle the LED on GP1 whenever the pushbutton is pressed.



In the main program, having configured Timer0 and selected the appropriate edge (falling) for the external interrupt, we need to enable both interrupt sources (as well as global interrupts):

```

; enable interrupts
movlw  1<<GIE|1<<T0IE|1<<INTE ; enable external, Timer0
movwf  INTCON                    ; and global interrupts

```

The interrupt service routine must include code to service both types of interrupt, but first we need to determine which source has triggered this interrupt – and that can be done by testing the various interrupt flags, as follows:

```

; *** Identify interrupt source
btfsc  INTCON,INTF      ; external
goto   ext_int
btfsc  INTCON,T0IF     ; Timer0
goto   t0_int
goto   isr_end         ; none of the above, so exit

```

The order is important, because it is possible that more than one interrupt source has triggered – that is, more than one of these flags may be set. That’s possible because more than one interrupt-triggering event, such as a timer overflow or an external signal, may have occurred while interrupts were disabled (for example, while another interrupt was being serviced).

So, if some interrupt sources (such as external events) are more important than others (such as timer overflows), you should structure your ISR so that the highest-priority interrupt sources are serviced first.

Note that the last instruction, ‘goto isr_end’, should never be executed. It is there to handle the case where an interrupt is triggered by a source that you haven’t written a handler for. If your hardware has a means of logging or informing the user of an error condition, you could use that capability here. Or it might be safest to reset your hardware, because clearly something has gone wrong! In this example, we just ignore the problem by immediately exiting the ISR. If you’re sure that nothing can ever go wrong, you could leave out this “catch all” goto.

The individual interrupt handlers are the same as before, except that they must finish with an instruction that skips to the end of the ISR, so that the other handlers are not executed.

For example:

```
ext_int ; *** Service external interrupt
;
;   Triggered on high -> low transition on INT pin
;   caused by externally debounced pushbutton press
;
;   Toggles LED on every high -> low transition
;
bcf     INTCON,INTF           ; clear interrupt flag

; toggle LED
movlw   1<<nB_LED             ; toggle indicator LED
xorwf   sGPIO,f              ; using shadow register
goto    isr_end
```

Of course, the handler immediately preceding the end of the ISR doesn’t need this ‘goto isr_end’ instruction, since it is at the end of the ISR anyway, but it’s a good idea to include it regardless, because it makes it easier to add more interrupt handlers later, without having to remember to add this ‘goto’.

Complete program

Here is the complete “toggle LED via external interrupt while flashing LED via timer interrupt” program, so that you can see how it all fits together:

```
*****
;   Description:      Lesson 6 example 6
;
;   Demonstrates handling of multiple interrupt sources
;
;   Toggles an LED when pushbutton on INT is pressed
;   (high -> low transition triggering external interrupt)
;   while another LED flashes at 1 Hz (driven by Timer0 interrupt)
;
*****
;
;   Pin assignments:
;   GP0 = flashing LED
;   GP1 = "button pressed" indicator LED
;   INT = pushbutton (active low)
;
*****
```

```

list      p=12F629
#include   <p12F629.inc>

errorlevel -302    ; no "register not in bank 0" warnings
errorlevel -312    ; no "page or bank selection not needed" messages

;***** CONFIGURATION
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
__CONFIG      _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

; pin assignments
constant      nF_LED=0            ; flashing LED on GP0
constant      nB_LED=1            ; "button pressed" indicator LED on GP1

;***** VARIABLE DEFINITIONS
CONTEXT       UDATA_SHR           ; variables used for context saving
cs_W          res 1
cs_STATUS     res 1

GENVAR        UDATA_SHR           ; general variables
sGPIO         res 1               ; shadow copy of GPIO
cnt_t0        res 1               ; counts timer0 interrupts
                ; (decremented by ISR every 250 us)
cnt_5ms       res 1               ; counts 5 ms periods
                ; (decremented by ISR every 5 ms)

;***** RESET VECTOR *****
RESET         CODE    0x0000       ; processor reset vector
                pagesel start
                goto     start

;***** INTERRUPT SERVICE ROUTINE *****
ISR           CODE    0x0004
                ; *** Save context
                movwf   cs_W         ; save W
                movf    STATUS,w     ; save STATUS
                movwf   cs_STATUS

                ; *** Identify interrupt source
                btfsc   INTCON,INTF   ; external
                goto   ext_int
                btfsc   INTCON,T0IF   ; Timer0
                goto   t0_int
                goto   isr_end        ; none of the above, so exit

ext_int ; *** Service external interrupt
                ;
                ; Triggered on high -> low transition on INT pin
                ; caused by externally debounced pushbutton press
                ;
                ; Toggles LED on every high -> low transition
                ;
                bcf     INTCON,INTF    ; clear interrupt flag

```

```

; toggle LED
movlw 1<<nB_LED          ; toggle indicator LED
xorwf sGPIO,f           ; using shadow register
goto isr_end

t0_int ; *** Service Timer0 interrupt
;
; TMR0 overflows every 250 clocks = 250 us
;
; Flashes LED at 1 Hz by toggling every 500 ms
; (every 250th 2 ms period)
;
movlw .256-.250+.3      ; add value to Timer0
banksel TMR0            ; for overflow after 250 counts
addwf TMR0,f
bcf INTCON,T0IF        ; clear interrupt flag

; count interrupts to generate 5 ms tick
decfsz cnt_t0,f        ; decrement interrupt count
goto isr_end           ; when count = 0
movlw .5000/.250       ; reload count for next 5 ms period
movwf cnt_t0           ; (5ms / 250us/interrupt)

; toggle flashing LED every 500 ms
decfsz cnt_5ms,f       ; decrement 5 ms tick count
goto flash_end         ; when count = 0
movlw .500/.5           ; reload count for next 500 ms period
movwf cnt_5ms          ; (500ms / 2ms/tick)

movf sGPIO,w           ; toggle LED
xorlw 1<<nF_LED         ; using shadow register
movwf sGPIO

flash_end
goto isr_end

isr_end ; *** Restore context then return
movf cs_STATUS,w       ; restore STATUS
movwf STATUS
swapf cs_W,f           ; restore W
swapf cs_W,w
retfie

;***** MAIN PROGRAM *****
MAIN CODE
start ; calibrate internal RC oscillator
call 0x03FF            ; retrieve factory calibration value
banksel OSCCAL         ; (stored at 0x3FF as a retlw k)
movwf OSCCAL           ; then update OSCCAL

;***** Initialisation

; configure port
banksel GPIO
clrf GPIO              ; start with all LEDs off
clrf sGPIO             ; update shadow
movlw ~(1<<nB_LED|1<<nF_LED) ; configure LED pins as outputs
banksel TRISIO
movwf TRISIO

```

```

; configure timer
movlw    b'11001000'    ; configure Timer0:
                    ; --0-----    timer mode (T0CS = 0)
                    ; ----1---    no prescaling (PSA = 1)
                    ;             ; (prescaler assigned to WDT)
banksel  OPTION_REG    ; -> increment TMR0 every 1 us
movwf    OPTION_REG

; initialise variables
movlw    .5000/.250    ; timer0 overflow count = 5ms / 250us/overflow
movwf    cnt_t0        ; (-> 5 ms per tick)
movlw    .500/.5       ; 5 ms tick count = 500ms / 5ms
movwf    cnt_5ms      ; (-> toggle LED every 500 ms)

; configure external interrupt
banksel  OPTION_REG
bcf      OPTION_REG,INTEDG ; trigger on falling edge (INTEDG = 0)

; enable interrupts
movlw    1<<GIE|1<<T0IE|1<<INTE ; enable external, Timer0
movwf    INTCON        ; and global interrupts

;***** Main loop
main_loop
; continually copy shadow GPIO to port
movf    sGPIO,w
banksel GPIO
movwf   GPIO

; repeat forever
goto    main_loop

END

```

Conclusion

Although this lesson has barely scratched the surface of what can be done with interrupts on mid-range PICs, we've seen, especially in examples 4 and 6, how interrupts make it possible to maintain background tasks (such as flashing an LED), while responding to and processing events (such as detecting and debouncing key presses), in a way that would be much more difficult to achieve if interrupts were not available.

We'll see more examples as topics are introduced in future lessons.

The next interrupt source we'll look at is "interrupt on change", which is commonly used to wake the PIC from sleep mode. It is covered in the [next lesson](#), along with the watchdog timer.

Introduction to PIC Programming

Mid-Range Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 7: Interrupt-on-change, Sleep Mode and the Watchdog Timer

One of the most useful features of modern microcontrollers (including PICs) is their ability to enter a power-saving “sleep” mode, where power drain may be less than a microwatt, facilitating the design of low-powered devices without traditional on-off switches – the device can turn itself “off”. For example, the [Gooligum Christmas Star](#), based on a PIC12F683, runs on a pair of N-cell batteries, but will remain “shut off”, with no significant battery drain, for a year or more, coming to life as soon as a pushbutton is pressed.

The latter feature relies on the PIC’s “interrupt-on-change” facility, which is often used to wake the device from sleep. As we shall see, it can also (as the name suggests) be used to trigger an interrupt in response to a changing input; similar to the external interrupt facility introduced in [lesson 6](#).

Another facility usually found in modern microcontrollers (including PICs) is a “watchdog timer”, intended to make a device more robust by providing a means of detecting situations where the program appears to be hung, and then resetting the processor so that the system can recover.

But as we’ll see in this lesson, the watchdog timer can also be used to periodically wake the PIC from sleep, a facility which makes it possible to design devices which spend most of their time sleeping, drawing very little current (and hence power) on average.

In summary, this lesson covers:

- Interrupt-on-change
- Sleep mode (power down)
- Wake-up on change (power up)
- The watchdog timer
- Periodic wake from sleep

Interrupt-on-change

In [lesson 6](#), we saw that the 12F629’s external interrupt facility can be used to trigger an interrupt on each rising or falling transition on the INT (GP2) pin; useful when we need to respond to an external digital signal more quickly than using a timer interrupt to poll the input every millisecond or so, without having to tie up the processor with a tight polling loop. Instead, the processor can be going about other tasks, but still be able to service the external event within microseconds of it occurring.

Note that the external interrupt is triggered on either a rising or falling edge (selectable by the INTEDG bit), but not both. If rising edges are selected, falling edges will be ignored. This tends to simplify code, since we are normally only interested in one type of transition.

The mid-range PIC architecture only supports a single external interrupt pin. However, the interrupt-on-change facility can be used if you need to respond quickly to a number of digital signal sources.

GP port change interrupts are enabled by setting the GPIE bit in the INTCON register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INTCON	GIE	PEIE	T0IE	INTE	GPIE	T0IF	INTF	GPIF

As always, for any interrupts to occur, the global interrupt enable bit, GIE, must also be set.

Every pin in the GPIO port can be enabled independently for interrupt-on-change.

This is controlled by the IOC register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IOC	-	-	IOC5	IOC4	IOC3	IOC2	IOC1	IOC0

If a bit in the IOC register is set, the corresponding GPIO pin will be enabled for interrupt-on-change.

For example, to enable interrupt-on-change for GP2, we would set IOC2 = 1.

If a pin is enabled for interrupt on change, any change in the state of that pin (since the last time the port was read or written) will create a *mismatch condition* and set the GPIF flag in the INTCON register. If GP port change interrupts are also enabled (GPIE = 1 and GIE = 1), an interrupt will be triggered.

This means that you should read or write GPIO immediately before enabling the port change interrupt, to end any existing mismatch condition, avoiding the interrupt being triggered the moment it is enabled. Similarly, in the interrupt service routine it is important to read or write GPIO, to end the mismatch before clearing the GPIF flag. If you do not end the mismatch, you will not be able to clear GPIF, and the interrupt will re-trigger, as soon as the ISR ends.

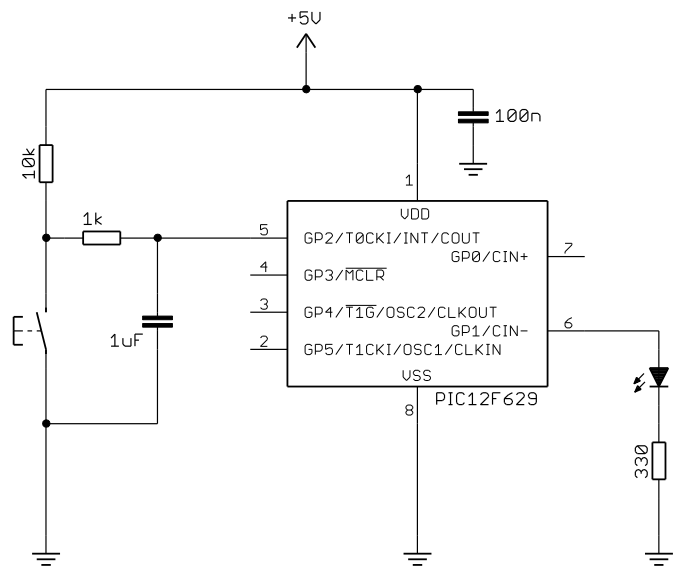
Note also that, unlike external interrupts, any change – whether a rising or falling transition – will trigger a port change interrupt.

Example 1: Interrupt-on-change (single input)

We'll start by demonstrating how to use interrupt-on-change to respond to a single input, using the circuit from the external interrupt example in [lesson 6](#) (shown on the right), where a pushbutton is connected to GP2 via a simple RC filter.

If you have the [Gooligum training board](#), close jumpers JP3, JP7 and JP12 to enable the 10 kΩ pull-up resistors on \overline{MCLR} (not shown here) and GP2 and the LED on GP1.

You must also add a 1 μF capacitor (supplied with the board) between GP2 and ground. You can do via pins 13 ('GP/RA/RB2') and 16 ('GND') on the 16-pin expansion header. There should be no need to use the solderless breadboard – simply plug the capacitor directly into these header pins.



As we did in that example, we'll toggle the LED on GP1 whenever the pushbutton is pressed.

GP2 was used because, on the 12F629, it has a Schmitt-trigger input, allowing the simple RC filter to provide effective hardware debouncing, as explained in [baseline lesson 4](#).

This is necessary because, although the switch debouncing could be implemented in software, it is difficult to do so while responding quickly to changes.

Firstly, in our initialisation code, after configuring and initialising GPIO as usual, we need to enable interrupt-on-change on GP2:

```

; configure port
banksel GPIO
clrf GPIO ; start with LED off
clrf sGPIO ; update shadow
movlw ~(1<<nB_LED) ; configure LED pin (only) as an output
banksel TRISIO
movwf TRISIO
banksel IOC ; enable interrupt-on-change
bsf IOC,nBUTTON ; on pushbutton input

```

(where 'nBUTTON' is a constant which has been set to '2')

The initial write to GPIO will have cleared any existing port mismatch condition, so it is safe to enable port change interrupts;:

```

; enable interrupts
movlw 1<<GIE|1<<GPIE ; enable port change and global interrupts
movwf INTCON

```

In the interrupt handler, we must read GPIO to clear the port mismatch condition which triggered this interrupt and (as for all interrupts) clear the interrupt flag:

```

banksel GPIO
movf GPIO,w ; clear mismatch condition
bcf INTCON,GPIF ; clear interrupt flag

```

Since the port change interrupt is triggered by any change, the ISR will be run on both button press and button release. This is different from the external interrupt example in [lesson 6](#), where the ISR only had to handle button press events.

Therefore, we must check whether the button had been pressed or released:

```

; toggle LED only on button press
btfsc GPIO,nBUTTON ; is button down?
goto isr_end

```

If the button was pressed, we can toggle the LED on GP1, as we've done before:

```

movlw 1<<nB_LED ; if so, toggle indicator LED
xorwf sGPIO,f ; using shadow register

```

(where 'nB_LED' is a constant which has been set to '1')

Otherwise, the code, including processor context save and restore, is essentially the same as that in the examples from [lesson 6](#).

Complete program

Here is how these pieces fit together, along with interrupt code framework introduced in [lesson 6](#):

```

;*****
;
; Description: Lesson 7 example 1
;
; Demonstrates use of interrupt-on-change interrupts
; (without software debouncing)
;
; Toggles LED when pushbutton is pressed (high -> low transition)
;
;*****
;
; Pin assignments:
; GP1 = indicator LED
; GP2 = pushbutton (externally debounced, active low)
;
;*****

list p=12F629
#include <p12F629.inc>

errorlevel -302 ; no "register not in bank 0" warnings
errorlevel -312 ; no "page or bank selection not needed" messages

;***** CONFIGURATION
; ext reset, no code or data protect, no brownout detect,
; no watchdog, power-up timer, 4Mhz int clock
__CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

; pin assignments
constant nB_LED=1 ; "button pressed" indicator LED on GP1
constant nBUTTON=2 ; externally debounced pushbutton on GP2

;***** VARIABLE DEFINITIONS
CONTEXT UDATA_SHR ; variables used for context saving
cs_W res 1
cs_STATUS res 1

GENVAR UDATA_SHR ; general variables
sGPIO res 1 ; shadow copy of GPIO

;***** RESET VECTOR *****
RESET CODE 0x0000 ; processor reset vector
pagesel start
goto start

;***** INTERRUPT SERVICE ROUTINE *****
ISR CODE 0x0004
; *** Save context
movwf cs_W ; save W
movf STATUS,w ; save STATUS
movwf cs_STATUS

; *** Service port change interrupt

```

```

;   Triggered on any transition on IOC-enabled input pin
;   caused by externally debounced pushbutton press
;
;   Toggles LED on every high -> low transition
;
;   (only port change interrupts are enabled)
;
banksel GPIO
movf    GPIO,w           ; clear mismatch condition
bcf     INTCON,GPIF     ; clear interrupt flag

; toggle LED only on button press
btfsc  GPIO,nBUTTON    ; is button down?
goto   isr_end
movlw  1<<nB_LED       ; if so, toggle indicator LED
xorwf  sGPIO,f         ; using shadow register

isr_end ; *** Restore context then return
movf   cs_STATUS,w     ; restore STATUS
movwf  STATUS
swapf  cs_W,f         ; restore W
swapf  cs_W,w
retfie

;***** MAIN PROGRAM *****
MAIN   CODE
start  ; calibrate internal RC oscillator
      call    0x03FF           ; retrieve factory calibration value
      banksel OSCCAL          ; (stored at 0x3FF as a retlw k)
      movwf  OSCCAL          ; then update OSCCAL

;***** Initialisation

      ; configure port
      banksel GPIO
      clrf   GPIO             ; start with LED off
      clrf   sGPIO           ; update shadow
      movlw  ~(1<<nB_LED)     ; configure LED pin (only) as an output
      banksel TRISIO
      movwf  TRISIO
      banksel IOC             ; enable interrupt-on-change
      bsf    IOC,nBUTTON     ; on pushbutton input

      ; enable interrupts
      movlw  1<<GIE|1<<GPIE  ; enable port change and global interrupts
      movwf  INTCON

;***** Main loop
main_loop
      ; continually copy shadow GPIO to port
      movf   sGPIO,w
      banksel GPIO
      movwf  GPIO

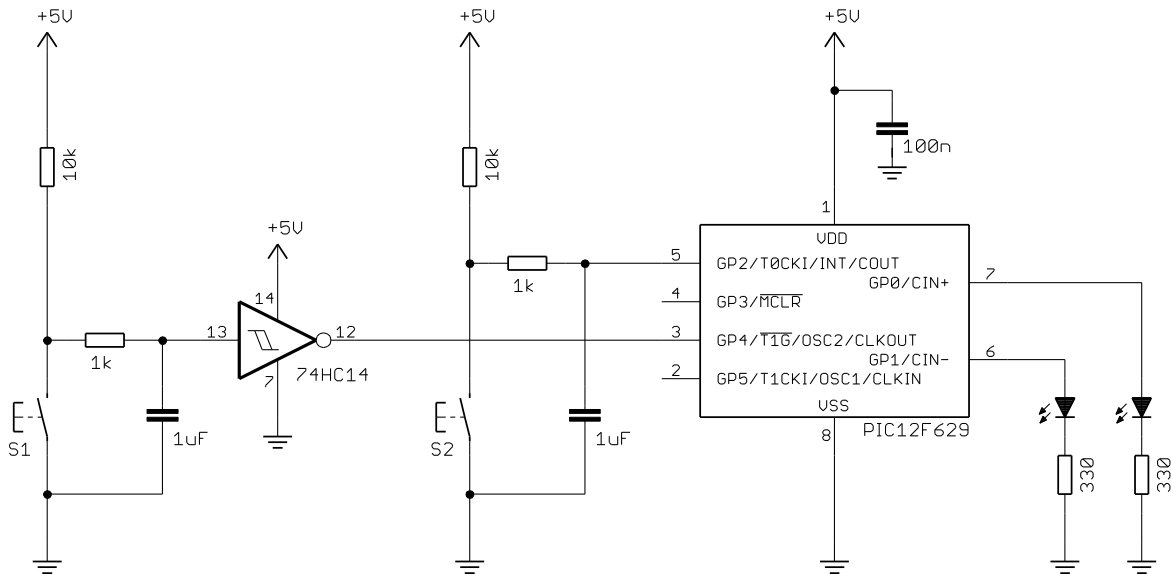
      ; repeat forever
      goto  main_loop

      END

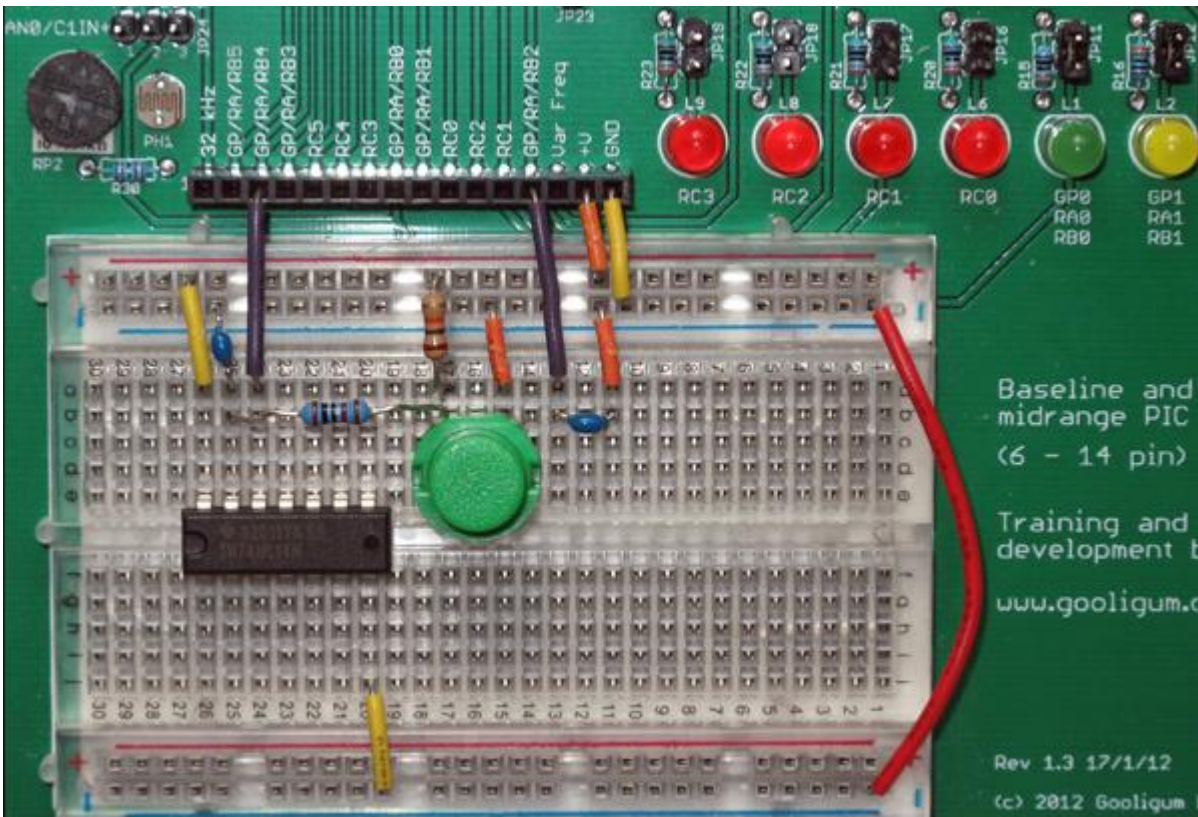
```

Example 2: Interrupt-on-change (multiple inputs)

This example demonstrates how to handle the situation where interrupt-on-change is enabled on more than one input pin, using the circuit (with the reset switch and pull-up omitted for clarity) shown below:



You can build this circuit with the [Gooligum training board](#), using the supplied 74HC14 Schmitt-trigger inverter, 1 kΩ and 10 kΩ resistors, 1 μF capacitors and pushbutton switch – connecting them to signals on the 16-pin header: GP4 input on pin 3 ('GP/RA/RB4'), GP2 input on pin 13 ('GP/RA/RB2') and ground and +5 V on pins 15 ('+V') and 16 ('GND') – using the solderless breadboard, as illustrated below:



You should also close JP3, JP7, JP11 and JP12 to enable the pull-up resistors on $\overline{\text{MCLR}}$ (not shown here) and GP2 and the LEDs on GP0 and GP1.

If you are using Microchip's Low Pin Count Demo Board, you can build the circuit in a similar way, by making connections to the 14-pin header on that board, although of course you will have to supply your own components and breadboard.

Each pushbutton toggles an LED: S1 controls the LED on GP1, and S2 controls the LED on GP0.

Once again, both buttons are debounced using hardware, to avoid messy software debounce routines (if we were going to implement software debouncing, we'd be better off using a timer interrupt to poll the inputs, as we did in [lesson 6](#)). For effective hardware debouncing, the simple RC filters need to be coupled with Schmitt-trigger inputs, and since the only available Schmitt-trigger GP input on the 12F629 is GP2, an external Schmitt-trigger inverter is used to drive GP4.

Thus, the operation of S1 is inverted, with respect to S2; GP4 is driven high when S1 is pressed, while GP2 is pulled low when S2 is pressed. We will have to take this difference into account.

The basic difficulty with having interrupt-on-change enabled for more than one input is that there are no flags to indicate which input changed; the GPIF flag can tell you that a port change has happened, but not which pin changed.

So when a port change interrupt occurs, we need to deduce which pin(s) have changed, by reading GPIO and comparing it to the last recorded state. And then, before exiting the ISR, we need to update our "last state" record, ready for next time.

Hence, we need some variables to store this information:

```
GENVAR      UDATA_SHR      ; general variables
sGPIO       res 1          ; shadow copy of GPIO
lGPIO       res 1          ; last state of GPIO (for change detection)
cGPIO       res 1          ; current state of GPIO (used by IOC ISR)
```

In the initialisation code, we need to enable interrupt-on-change for both inputs, and update the "last state" variable, so that everything is in sync:

```
banksel GPIO
movf GPIO,w          ; update last port state
movwf lGPIO         ; (for change detection)
banksel IOC          ; enable interrupt-on-change
movlw 1<<nPB1|1<<nPB2 ; on pushbuttons 1 and 2
movwf IOC
```

Then, when handling the port change interrupt in the ISR, we need to determine which pins have changed. This can be done by XORing the current state of GPIO with the last recorded state. Since an XOR operation only results in a '1' where the inputs differ, this is a means of detecting which bits have changed:

```
bcf INTCON,GPIF      ; clear interrupt flag

; determine which pins have changed
banksel GPIO         ; read GPIO
movf GPIO,w          ; to clear mismatch condition
movwf cGPIO          ; and save current state
xorwf lGPIO,f        ; XOR with last state to detect changes
```

Note that the result of the XOR was written back to 1GPIO, which now contains ‘0’s in bit positions where the current state matches the last state and ‘1’s where they differ. That is, if a pin has changed, the corresponding bit in 1GPIO will be set to ‘1’.

Next we need to check each bit in 1GPIO corresponding to the interrupt-on-change inputs, and toggle the appropriate LED if that input has changed:

```

        ; toggle LED 1 only on button 1 press (active low)
        btfss    1GPIO,nPB1          ; has button 1 changed?
        goto     ioc_pb2             ; check next button if not
        btfsc    cGPIO,nPB1         ; is button down (low)?
        goto     ioc_pb2             ; check next button if not
        movlw    1<<nB1_LED          ; if so, toggle LED 1
        xorwf    sGPIO,f             ; using shadow register

ioc_pb2 ; toggle LED 2 only on button 2 press (active high)
        btfss    1GPIO,nPB2          ; has button 2 changed?
        goto     ioc_end             ; finish IOC if not
        btfss    cGPIO,nPB2         ; is button down (high)?
        goto     ioc_end             ; finish IOC if not
        movlw    1<<nB2_LED          ; if so, toggle LED 2
        xorwf    sGPIO,f             ; using shadow register

```

Note that the test for “button press” button 2 is opposite to that for button 1, because of the external inverter on the GP4 input, as discussed above.

Finally, we need to record the current GPIO state, for reference as the “last state”, the next time a port change interrupt occurs:

```

ioc_end ; update last GPIO state (for next time)
        movf     cGPIO,w             ; copy current state of GPIO
        movwf    1GPIO              ; to last state

```

Complete program

Here is how these pieces fit into the framework used in the first example, to form the complete “interrupt-on-change with multiple inputs” program:

```

;*****
; Description:      Lesson 7 example 2
;
; Demonstrates handling of multiple interrupt-on-change interrupts
; (without software debouncing)
;
; Toggles LED on GP0 when pushbutton on GP2 is pressed
; (high -> low transition)
; and LED on GP1 when pushbutton on GP4 is pressed
; (low -> high transition)
;
;*****
;
; Pin assignments:
; GP0 = indicator LED 1
; GP1 = indicator LED 2
; GP2 = pushbutton 1 (externally debounced, active low)
; GP4 = pushbutton 2 (externally debounced, active high)
;
;*****

```

```

list          p=12F629
#include      <p12F629.inc>

errorlevel   -302      ; no "register not in bank 0" warnings
errorlevel   -312      ; no "page or bank selection not needed" messages

;***** CONFIGURATION
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
__CONFIG     _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

; pin assignments
constant     nB1_LED=0          ; "button 1 pressed" indicator LED on GP0
constant     nB2_LED=1          ; "button 2 pressed" indicator LED on GP1
constant     nPB1=2             ; button 1 (ext debounce, active low) on GP2
constant     nPB2=4             ; button 2 (ext debounce, active high) on GP4

;***** VARIABLE DEFINITIONS
CONTEXT      UDATA_SHR          ; variables used for context saving
cs_W        res 1
cs_STATUS    res 1

GENVAR       UDATA_SHR          ; general variables
sGPIO       res 1              ; shadow copy of GPIO
lGPIO       res 1              ; last state of GPIO (for change detection)
cGPIO       res 1              ; current state of GPIO (used by IOC ISR)

;***** RESET VECTOR *****
RESET       CODE    0x0000      ; processor reset vector
           pagesel start
           goto     start

;***** INTERRUPT SERVICE ROUTINE *****
ISR         CODE    0x0004
           ; *** Save context
           movwf   cs_W          ; save W
           movf    STATUS,w      ; save STATUS
           movwf   cs_STATUS

           ; *** Service port change interrupt
           ;
           ;   Triggered on any transition on IOC-enabled input pins
           ;   caused by externally debounced pushbutton press
           ;
           ;   Toggles LED1 on every high -> low transition of PB1
           ;   and LED2 on every low -> high transition of PB2
           ;
           ;   (only port change interrupts are enabled)
           ;
           bcf     INTCON,GPIF    ; clear interrupt flag

           ; determine which pins have changed
           banksel GPIO          ; read GPIO
           movf   GPIO,w         ; to clear mismatch condition
           movwf  cGPIO         ; and save current state
           xorwf  lGPIO,f       ; XOR with last state to detect changes

```

```

; toggle LED 1 only on button 1 press (active low)
btfss  lGPIO,nPB1          ; has button 1 changed?
goto   ioc_pb2             ; check next button if not
btfsc  cGPIO,nPB1         ; is button down (low)?
goto   ioc_pb2             ; check next button if not
movlw  1<<nB1_LED          ; if so, toggle LED 1
xorwf  sGPIO,f            ; using shadow register

ioc_pb2 ; toggle LED 2 only on button 2 press (active high)
btfss  lGPIO,nPB2         ; has button 2 changed?
goto   ioc_end             ; finish IOC if not
btfsc  cGPIO,nPB2         ; is button down (high)?
goto   ioc_end             ; finish IOC if not
movlw  1<<nB2_LED          ; if so, toggle LED 2
xorwf  sGPIO,f            ; using shadow register

ioc_end ; update last GPIO state (for next time)
movf   cGPIO,w            ; copy current state of GPIO
movwf  lGPIO              ; to last state

isr_end ; *** Restore context then return
movf   cs_STATUS,w       ; restore STATUS
movwf  STATUS
swapf  cs_W,f            ; restore W
swapf  cs_W,w
retfie

;***** MAIN PROGRAM *****
MAIN    CODE
start  ; calibrate internal RC oscillator
      call  0x03FF          ; retrieve factory calibration value
      banksel OSCCAL        ; (stored at 0x3FF as a retlw k)
      movwf OSCCAL         ; then update OSCCAL

;***** Initialisation

      ; configure port
      banksel GPIO
      clrf  GPIO           ; start with LED off
      clrf  sGPIO          ; update shadow
      movlw ~(1<<nB1_LED|1<<nB2_LED) ; configure LED pins as outputs
      banksel TRISIO
      movwf TRISIO
      banksel GPIO
      movf  GPIO,w         ; update last port state
      movwf lGPIO          ; (for change detection)
      banksel IOC          ; enable interrupt-on-change
      movlw 1<<nPB1|1<<nPB2 ; on pushbuttons 1 and 2
      movwf IOC

      ; enable interrupts
      movlw 1<<GIE|1<<GPIE ; enable port change and global interrupts
      movwf INTCON

;***** Main loop
main_loop
      ; continually copy shadow GPIO to port
      movf  sGPIO,w

```



```

banksel GPIO
movwf GPIO

; repeat forever
goto main_loop

END

```

Sleep Mode

As mentioned earlier, mid-range PICs are able to enter a standby, or *sleep* mode, to save power.

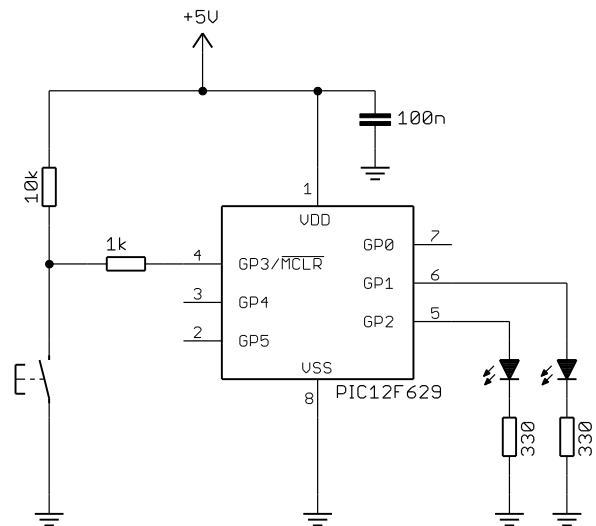
In this mode, the PIC12F629 will typically draw less than 3 nA (down to only 1 nA when the power supply is reduced to 2 V), when all of the power-consuming facilities (such as the watchdog timer; see later) have been disabled and the output pins are not supplying any current.

To demonstrate how it is used, we'll use the circuit from [lesson 6](#), shown on the right.

It consists of a PIC12F629, LEDs on GP1 and GP2, and a pushbutton switch on GP3.

If you have the [Gooligum training board](#), close jumpers JP3, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP1 and GP2. Or, if you are using Microchip's Low Pin Count Demo Board, you will need to connect LEDs to GP1 and GP2, as described in [baseline lesson 1](#).

To demonstrate to yourself that power consumption really is reduced when the PIC enters sleep mode, you would have to use an external power supply, instead of using your PICkit 2 or PICkit 3 to power the circuit. You can then place a multimeter in-line with the power supply, to measure the supply current.



The instruction for placing the PIC into standby mode is 'sleep' – "enter **sleep** mode".

To illustrate the use of the sleep instruction, consider the following fragment of code. It turns on the LED on GP1, waits for the button to be pressed, and then enters sleep mode:

```

movlw    ~(1<<GP1)           ; configure LED pin as output
banksel TRISIO
movwf   TRISIO

banksel GPIO
bsf     GPIO,GP1             ; turn on LED

waitlo  btfsc  GPIO,GP3      ; wait for button press (low)
        goto   waitlo

sleep   ; enter sleep mode

goto   $                    ; (this instruction should never run)

```

Note that the final ‘goto \$’ instruction (an endless loop) will never be executed, because ‘sleep’ will halt the processor; any instructions after ‘sleep’ will never be reached.

When you run this program, the LED will turn on and then, when you press the button, nothing will appear to happen! The LED stays on. Shouldn’t it turn off? What’s going on?

The current supplied from a 5 V supply, before pressing the button, with the LED on, was measured¹ to be 8.58 mA. After pressing the button, the measured current dropped to 7.86 mA, a fall of only 0.72 mA.

This happens because, when the PIC goes into standby mode, it stops executing instructions, saving some power ($0.70 \text{ mA} \times 5 \text{ V} = 3.5 \text{ mW}$ in this case), but the I/O ports remain in the state they were in, before the ‘sleep’ instruction was executed.

Note: For low power consumption in standby mode, the I/O ports must be configured to stop sourcing or sinking current, before entering SLEEP mode.

In this case, the fix is simple – turn off the LED before entering sleep mode, as follows:

```

movlw    ~(1<<GP1)           ; configure LED pin as output
banksel  TRISIO
movwf    TRISIO

banksel  GPIO
bsf      GPIO,GP1           ; turn on LED

waitlo   btfsc   GPIO,GP3    ; wait for button press (low)
         goto    waitlo

         bcf      GPIO,GP1    ; turn off LED

         sleep                    ; enter sleep mode

         goto    $              ; (this instruction should never run)

```

When this program is run, the LED will turn off when the button is pressed.

The current measured² with the PIC in standby and the LED off was less than $0.1 \mu\text{A}$ – too low to register on the multimeter used! That was with the unused pins tied to VDD or VSS (whichever is most convenient on the circuit board), as floating CMOS inputs can lead to unnecessary current draw.

Note: To minimise power in standby mode, configure all unused pins as inputs, and tie them VDD or VSS through 10 kΩ resistors. Do not connect them directly to VDD or VSS, as the PIC may be damaged if these pins are inadvertently configured as outputs.

For clarity, tying the unused inputs to VDD or VSS was not shown in the circuit diagram above.

¹ With the minimal circuit built on solderless breadboard. If you leave the PIC in a development board, such as the Gooligum training board, other devices on the board may draw current, meaning that you are likely to see higher currents than this. You should nevertheless see a drop in current when the PIC enters sleep mode.

² Again, with the circuit built separately on a breadboard.

Wake-up from sleep

Sleep mode would not be useful if there was no way to wake up from it – there has to be a way to turn the device “on” when needed (perhaps in response to an event, such as a button press), after it has been turned “off”.

Mid-range PICs provide a number of ways to wake from sleep mode:

- Any device reset, such as an external reset signal on the $\overline{\text{MCLR}}$ pin (if enabled)
- Watchdog timer timeout (see the section on the watchdog timer, later in this lesson)
- Any enabled interrupt source which can set its interrupt flag while in sleep mode

Some interrupt sources cannot be used wake the device from sleep, because, in sleep mode, the PIC’s clock, or oscillator, is not running. For example, the Timer0 interrupt cannot be used for wake-up from sleep, because TMR0 does not increment while the PIC is in sleep mode.

However, external (INT pin) and port change interrupts can be used for wake-up on mid-range PICs, as well as some other interrupt sources, such as Timer1 and comparators, that we will examine in later lessons.

In this lesson, we’ll look at how to use the port change interrupt to wake a PIC from sleep mode; the method for using an external interrupt is essentially the same, but is of course limited to only the INT pin.

Example 4: Using interrupt-on-change for wake-up from sleep

In [baseline lesson 7](#), we saw that a “wake-up on change” facility is available in the baseline architecture on a handful of pins, but that it is an all or nothing affair; either all of the available pins are enabled for wake-up on change, or none of them are.

The mid-range equivalent to wake-up on change is the interrupt-on-change facility introduced above. It is more flexible, in that interrupt-on-change can be enabled independently on each pin. And on the 12F629, interrupt-on-change is available on every pin in GPIO.

“Interrupt-on-change” can be used to wake the device from sleep, even if interrupts are not enabled. If port change interrupts are enabled ($\text{GPIE} = 1$), but global interrupts are disabled ($\text{GIE} = 0$), then the device will wake from sleep when an IOC-enabled input changes, but no interrupt will occur. Program execution simply continues with the instruction following the `sleep` instruction.

Note: in the mid-range PIC architecture, on wake-up from sleep, program execution continues with the instruction following `sleep`. No device reset occurs (unless a reset event, such as a reset signal on $\overline{\text{MCLR}}$ caused the wake-up). This is different from the baseline architecture.

If global interrupts are enabled ($\text{GIE} = 1$) when the device wakes from sleep, the PIC will execute the instruction following `sleep`, and then enter the interrupt service routine.

If you want the PIC to execute the ISR immediately after it wakes from sleep, you need to enable interrupts and place a `nop` (“do nothing”) instruction immediately following the `sleep` instruction.

If you are using other interrupts in your program, and don’t want to execute the ISR when the PIC wakes from sleep, simply disable interrupts (clear GIE) before entering sleep mode.

But regardless of whether interrupts are enabled or not, if $\text{GPIE} = 1$, the PIC will wake when the value of any IOC-enabled input changes while it is in sleep mode.

It is important to clear the GPIF flag before entering sleep mode, or else the PIC will wake immediately.

Note: You should read the input pins configured for interrupt-on-change just prior to entering sleep mode, and clear GPIF. Otherwise, if the value at an IOC-enabled pin had changed since the last time it was read, the PIC will wake immediately upon entering sleep mode, as the input value would be seen to be different from that last read.

It is also important to ensure that any input which will be used to trigger a wake-up is stable before entering sleep mode. Consider what would happen if interrupt-on-change was enabled in the program above. As soon as the button is pressed, the LED will turn off and the PIC will enter standby mode, as intended. But on the first switch bounce, the input would be seen to have changed, and the PIC would wake.

Even if the circuit included hardware debouncing, there's still a problem: the LED will go off and the PIC will enter standby as soon as the button is pressed, but when the button is subsequently released, it will be seen as a change, and the PIC will wake up! To successfully use the pushbutton to turn the circuit (PIC and LED) "off", it is necessary to wait for the button to be released and remain stable (debounced) before entering sleep mode.

But there's still a potential problem. Assume that, in this example, we want to wake-up the PIC and turn the LED on when the button is pressed. PICs are fast, and human fingers are slow – if, as soon as the PIC waits from sleep, the program immediately checks for a "turn off" button press, the button will still be down, as part of the button press which woke the PIC from sleep, and the LED will immediately turn off again. To avoid this, we must wait for the button to be in a stable "up" state before checking that it is "down".

So the necessary sequence is:

```
loop
  turn on LED
  wait for stable button high
  wait for button low
  turn off LED
  wait for stable button high
  clear GPIF
  sleep
  goto loop ; repeat from the beginning
```

The following code, which makes use of the debounce macro defined in [lesson 5](#), implements this:

```
***** Initialisation
start
  ; configure port
  movlw  ~(1<<nLED)      ; configure LED pin as output
  banksel TRISIO
  movwf  TRISIO

  ; configure interrupt-on-change
  banksel IOC           ; enable interrupt-on-change
  bsf    IOC,nBUTTON    ; on pushbutton input
  bsf    INTCON,GPIE    ; enable wake-up (interrupt) on port change

  ; configure Timer0 (for DbnceHi macro)
  movlw  b'11000111'    ; configure Timer0:
  ; --0-----          timer mode (T0CS = 0)
  ; ----0----          prescaler assigned to Timer0 (PSA = 0)
  ; -----111         prescale = 256 (PS = 111)
  banksel OPTION_REG    ; -> increment TMR0 every 256 us
  movwf  OPTION_REG
```

```

;***** Main loop
main_loop
    ; turn on LED
    banksel GPIO
    bsf      LED

    ; wait for stable button high (in case it is still bouncing)
    DbnceHi BUTTON

    ; wait for button press
wait_lo  btfsc  BUTTON      ; wait until button low
        goto  wait_lo

    ; go into standby (low power) mode
    bcf      LED           ; turn off LED
    DbnceHi  BUTTON       ; wait for stable button release
    bcf      INTCON,GPIF   ; clear port change flag
    sleep                    ; enter sleep mode

    ; repeat forever
    goto    main_loop

```

(the labels 'LED', 'nLED', 'BUTTON' and 'nBUTTON' are defined earlier in the program)

This code does essentially the same thing as the “toggle an LED” programs developed in [lesson 3](#), except that in this case, when the LED is off, the PIC is drawing negligible power.

Watchdog Timer

In the real world, computer programs sometimes “crash”; they will stop responding to input, stuck in a continuous loop they can't get out of, and the only way out is to reset the processor (e.g. Ctrl-Alt-Del on Windows PCs – and even that sometimes won't work, and you need to power cycle a PC to bring it back). Microcontrollers are not immune to this. Their programs can become stuck because some unforeseen sequence of inputs has occurred, or perhaps because an expected input signal never arrives. Or, in the electrically noisy industrial environment in which microcontrollers are often operating, power glitches and EMI on signal lines can create an unstable environment, perhaps leading to a crash.

Crashes present a special problem for equipment which is intended to be reliable, operating autonomously, in environments where user intervention isn't an option.

One of the major functions of a *watchdog timer* is to automatically reset the microcontroller in the event of a crash. It is simply a free-running timer (running independently of any other processor function, including sleep) which, if allowed to overflow, will reset the PIC. In normal operation, an instruction which clears the watchdog timer is regularly executed – often enough to prevent the timer ever overflowing. This instruction is often placed in the “main loop” of a program, where it would normally be expected to be executed often enough to prevent watchdog timer overflows. If the program crashes, the main loop presumably won't complete; the watchdog timer won't be cleared, and the PIC will be reset. Hopefully, when the PIC restarts, whatever condition led to the crash will have gone away, and the PIC will resume normal operation.

The instruction for clearing the watchdog timer is 'clrwdt' – “**clear watchdog timer**”.

The watchdog timer has a nominal time-out period of 18 ms. If that's not long enough, it can be extended by using a prescaler.

As we saw in [lesson 4](#), a single prescaler is shared between Timer0 and the watchdog timer – it can be assigned to one or the other, but not both.

It is configured using a number of bits in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION_REG	GPPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0

To assign the prescaler to the watchdog timer, set the PSA bit to '1'.

When assigned to the watchdog timer, the prescale ratio is set by the PS<2:0> bits, as shown in the following table:

PS<2:0> bit value	WDT prescale ratio	WDT period (nominal)
000	1 : 1	18 ms
001	1 : 2	36 ms
010	1 : 4	72 ms
011	1 : 8	144 ms
100	1 : 16	288 ms
101	1 : 32	576 ms
110	1 : 64	1.15 s
111	1 : 128	2.30 s

Note that the prescale ratios are one half of those that apply when the prescaler is assigned to Timer0.

For example, if PSA = 1 (assigning the prescaler to the watchdog timer) and PS<2:0> = '011' (selecting a ratio of 1:8), the watchdog time-out period will be $8 \times 18 \text{ ms} = 144 \text{ ms}$.

With the maximum prescale ratio, the watchdog time-out period is $128 \times 18 \text{ ms} = 2.3 \text{ s}$.

The watchdog timer is controlled by the WDTE bit in the configuration word:

Bit 13	12	11	10	9	8	7	6	5	4	3	2	1	Bit 0
BG1	BG0	-	-	-	CPD	CP	BODEN	MCLRE	PWRTE	WDTE	FOSC2	FOSC1	FOSC0

Setting WDTE to '1' enables the watchdog timer.

To set WDTE, use the symbol '`_WDT_ON`' instead of '`_WDT_OFF`' in the `__CONFIG` directive.

Since the configuration word cannot be accessed by programs running on the PIC (it can only be written to when the PIC is being programmed), **the watchdog timer cannot be enabled or disabled at runtime**. It can only be configured to be 'on' or 'off' when the PIC is programmed.

Example 5a: Watchdog Timer

To show how the watchdog timer allows the PIC to recover from a crash, we'll use a simple program which turns on an LED for 1.0 sec, turns it off again, and then enters an endless loop (simulating a crash).

If the watchdog timer is disabled, the loop will never exit and the LED will remain off.

But if the watchdog timer is enabled, with a period of 2.3 sec, the program should restart itself after 2.3 sec, and the LED will flash: on for 1.0 sec and off for 1.3 sec (approximately).

To make it easy to test configurations with the watchdog timer on or off, you can use a construct such as:

```
#define WATCHDOG ; define to enable watchdog timer

IFDEF WATCHDOG
    ; ext reset, no code or data protect, no brownout detect,
    ; watchdog, power-up timer, 4 Mhz int clock
    __CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_ON &
    _PWRTE_ON & _INTRC_OSC_NOCLKOUT
ELSE
    ; ext reset, no code or data protect, no brownout detect,
    ; no watchdog, power-up timer, 4 Mhz int clock
    __CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
    _PWRTE_ON & _INTRC_OSC_NOCLKOUT
ENDIF
```

Note that these `__CONFIG` directives enable external reset (`'_MCLRE_ON'`), allowing the pushbutton switch connected to pin 4, to reset the PIC. That's useful because, with this program going into an endless loop, having to power cycle the PIC to restart it would be annoying; pressing the button is much more convenient.

The prescaler is set to 1:128 and assigned to the watchdog timer by:

```
movlw 1<<PSA | b'111' ; assign prescaler to WDT (PSA = 1)
                        ; prescale = 128 (PS = 111)
banksel OPTION_REG    ; -> WDT timeout = 2.3 s
movwf OPTION_REG
```

The code to flash the LED once and then enter an endless loop is simple, making use of the `'DelayMS'` macro introduced in [lesson 5](#):

```
banksel GPIO          ; turn on LED
bsf LED

DelayMS 1000         ; delay 1 sec

banksel GPIO          ; turn off LED
bcf LED

goto $               ; wait forever
```

Complete program

If you build and run this program, with `'#define WATCHDOG'` commented out (place a `';` in front of it), the LED will light once, and then remain off. But if you define `'WATCHDOG'`, the LED will continue to flash:

```
*****
;
; Description: Lesson 7, example 5a
;
; Demonstrates use of watchdog timer
;
; Turn on LED for 1 s, turn off, then enter endless loop
; LED stays off if watchdog not enabled, flashes if WDT set to 2.3 s
;
;*****
;
; Pin assignments:
; GP1 = indicator LED
;
;*****

list p=12F629
```

```

#include    <p12F629.inc>

#include    <stdmacros-mid.inc>          ; DelayMS - delay in milliseconds
                                           ; (calls delay10)
EXTERN    delay10                       ; W x 10ms delay

errorlevel -302      ; no "register not in bank 0" warnings
errorlevel -312      ; no "page or bank selection not needed" messages

radix      dec

;***** CONFIGURATION
#define     WATCHDOG           ; define to enable watchdog timer

IFDEF WATCHDOG
           ; ext reset, no code or data protect, no brownout detect,
           ; watchdog, power-up timer, 4 Mhz int clock
__CONFIG   _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_ON &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT
ELSE
           ; ext reset, no code or data protect, no brownout detect,
           ; no watchdog, power-up timer, 4 Mhz int clock
__CONFIG   _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT
ENDIF

; pin assignments
#define     LED      GPIO,1           ; indicator LED on GP1
constant   nLED=1                    ; (port bit 1)

;***** RESET VECTOR *****
RESET     CODE    0x0000           ; processor reset vector
           ; calibrate internal RC oscillator
           call   0x03FF           ; retrieve factory calibration value
           banksel OSCCAL           ; (stored at 0x3FF as a retlw k)
           movwf  OSCCAL           ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start
           ; configure port
           movlw  ~(1<<nLED)        ; configure LED pin as output
           banksel TRISIO
           movwf  TRISIO

           ; configure watchdog timer
           movlw  1<<(PSA | b'111' ; assign prescaler to WDT (PSA = 1)
                                           ; prescale = 128 (PS = 111)
           banksel OPTION_REG       ; -> WDT timeout = 2.3 s
           movwf  OPTION_REG

;***** Main code
           banksel GPIO             ; turn on LED
           bsf    LED

           DelayMS 1000             ; delay 1 sec

```



```

banksel GPIO          ; turn off LED
bcf    LED

goto   $              ; wait forever

END

```

Example 5b: Detecting a WDT time-out reset

When the watchdog timer times out, the PIC is reset, your program is restarted, in the same way that is was when power was first applied, or after an $\overline{\text{MCLR}}$ reset.

But you may want your program to behave differently, depending on why it was restarted. In particular, if a WDT time-out reset has occurred, you may wish to reset some external equipment to a known state, or perhaps simply turn on an alarm indicator to show that something has gone wrong.

Watchdog timer resets are indicated by the $\overline{\text{TO}}$ bit in the STATUS register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	IRP	RP1	RP0	$\overline{\text{TO}}$	$\overline{\text{PD}}$	Z	DC	C

The $\overline{\text{TO}}$ (time-out) bit is cleared to '0' by a WDT time-out reset.

It is set to '1' at power-on, or by entering sleep mode, or execution of the 'clrwdt' instruction.

Thus, if $\overline{\text{TO}}$ has been cleared, it means that a WDT time-out reset has occurred.

To demonstrate how the $\overline{\text{TO}}$ flag is used, the previous example can be modified, to light a second LED when a watchdog timer reset has occurred, but not when the PIC is first powered on, as follows:

```

;***** Initialisation
start
    ; configure port
    banksel GPIO          ; start with all LEDs off
    clrf    GPIO
    movlw   ~(1<<nLED|1<<nWDT) ; configure LED pins as outputs
    banksel TRISIO
    movwf   TRISIO

    ; configure watchdog timer
    movlw   1<<PSA | b'111' ; assign prescaler to WDT (PSA = 1)
                                ; prescale = 128 (PS = 111)
    banksel OPTION_REG    ; -> WDT timeout = 2.3 s
    movwf   OPTION_REG

;***** Main code
    ; test for WDT-timeout reset
    banksel GPIO
    btfss   STATUS,NOT_TO   ; if WDT timeout has occurred,
    bsf     WDT             ; turn on "error" LED

    ; flash LED
    bsf     LED             ; turn on "flash" LED
    DelayMS 1000           ; delay 1 sec

```

```

banksel GPIO                ; turn off "flash" LED
bcf     LED

; wait forever
goto   $

```

Note that, if, after the watchdog timer has reset the PIC, and the “WDT” LED has been lit, you use the reset button to restart the program, the “WDT” LED will remain lit. This is because a $\overline{\text{MCLR}}$ reset does not affect the $\overline{\text{TO}}$ bit.

Example 5c: Using the `clrwdt` instruction

Of course, you will normally want to avoid WDT time-out resets.

As discussed earlier, to prevent the watchdog timer timing out, simply place a ‘`clrwdt`’ instruction within the main loop.

A watchdog timer period should be selected which is long enough to ensure that the watchdog timer never expires within the loop, unless something is wrong. For example, if your main loop normally completes within 10 ms, but can sometimes take up to 40 ms, you would select a watchdog period of 72 ms (prescale ratio = 1:4) or perhaps 144 ms (prescale = 1:8) to be sure.

To demonstrate that the ‘`clrwdt`’ instruction really does stop the watchdog expiring (if executed often enough), simply include it in the endless loop at the end of the code:

```

loop   clrwdt                ; clear watchdog timer
      goto   loop            ; repeat forever

```

If you replace the ‘`goto $`’ line with this “clear watchdog timer” loop, you will find that, after flashing once, the LED will remain off – regardless of the watchdog timer setting.

Example 6: Periodic wake from sleep

The watchdog timer can also be used to wake the PIC from sleep mode.

This is useful in situations where inputs do not need to be responded to instantly, but can be checked periodically. To minimise power consumption, the PIC can sleep most of the time, waking up every so often (say, once per second), checking inputs and, if there is nothing to do, going back to sleep.

Note that a periodic wake-up can be combined with wake-up on pin change; you may for example wish to periodically log the value of a sensor, but also respond immediately to button presses.

If the watchdog timer expires while the PIC is in sleep mode, the device wakes from sleep, and program execution continues with the instruction following `sleep`. No device reset occurs³.

The `sleep` instruction clears the watchdog timer and prescaler. This means that the device will sleep for however long the watchdog timer period is set to (unless another event wakes it before the WDT expires).

³ This differs from WDT wake from sleep in the baseline architecture

To demonstrate how this works, we can simply convert the main code in example 5a into a loop, incorporating a 'sleep' instruction:

```
main_loop
    banksel GPIO          ; turn on LED
    bsf      LED

    DelayMS 1000          ; delay 1 sec

    banksel GPIO          ; turn off LED
    bcf      LED

    sleep                ; enter sleep mode (until WDT time-out)

    goto     main_loop    ; repeat forever
```

If you enable the watchdog timer, you'll find that the LED turns on for 1 s, and is then off for around 2 s, before turning on again. And if you measure the current drawn by the PIC, you will find that very little power is consumed while the LED is off, because the PIC is in sleep mode.

On the other hand, if you disable the watchdog timer, the LED will turn on for 1 s, but then turn off forever, because, with the watchdog disabled, the PIC never wakes from sleep.

Complete program

Here is how this new main loop fits into the program presented in example 5a:

```
*****
;
; Description:      Lesson 7, example 6
;
; Demonstrates periodic wake from sleep, using the watchdog timer
;
; Turn on LED for 1 s, turn off, then sleep
; LED stays off if watchdog not enabled,
; flashes (1 s on, 2.3 s off) if WDT enabled
;
*****
;
; Pin assignments:
; GP1 = indicator LED
;
*****

list      p=12F629
#include   <p12F629.inc>

#include   <stdmacros-mid.inc>      ; DelayMS - delay in milliseconds
                                           ; (calls delay10)
EXTERN    delay10                  ; W x 10ms delay

errorlevel -302      ; no "register not in bank 0" warnings
errorlevel -312      ; no "page or bank selection not needed" messages

radix     dec

***** CONFIGURATION
#define    WATCHDOG          ; define to enable watchdog timer
```

```

IFDEF WATCHDOG
    ; ext reset, no code or data protect, no brownout detect,
    ; watchdog, power-up timer, 4 Mhz int clock
    __CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_ON &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT
ELSE
    ; ext reset, no code or data protect, no brownout detect,
    ; no watchdog, power-up timer, 4 Mhz int clock
    __CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT
ENDIF

; pin assignments
#define LED GPIO,1 ; indicator LED on GP1
constant nLED=1 ; (port bit 1)

;***** RESET VECTOR *****
RESET CODE 0x0000 ; processor reset vector
; calibrate internal RC oscillator
call 0x03FF ; retrieve factory calibration value
banksel OSCCAL ; (stored at 0x3FF as a retlw k)
movwf OSCCAL ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start
; configure port
movlw ~(1<<nLED) ; configure LED pin as output
banksel TRISIO
movwf TRISIO

; configure watchdog timer
movlw 1<<PSA | b'111' ; assign prescaler to WDT (PSA = 1)
; prescale = 128 (PS = 111)
banksel OPTION_REG ; -> WDT timeout = 2.3 s
movwf OPTION_REG

;***** Main loop
main_loop
banksel GPIO ; turn on LED
bsf LED

DelayMS 1000 ; delay 1 sec

banksel GPIO ; turn off LED
bcf LED

sleep ; enter sleep mode (until WDT time-out)

goto main_loop ; repeat forever

END

```

Conclusion

We've seen in this lesson that change on a digital input pin can be used to interrupt mid-range PICs, and that, with a little effort, we can determine which pin changed.

We also saw that mid-range PICs can be put into a low-power sleep mode, and that they can be made to wake up by an external event (such as a pin change), or on a regular basis by the watchdog timer, which is also (in fact, primarily) useful for restarting the device if it gets "stuck", following some type of error condition.

So far in this tutorial series we've focussed on programming and the internal architecture of mid-range PICs, but in the [next lesson](#) we'll dive into hardware, taking a look at features related to the power supply, such as brown-out detection and the power-up timer, and the available oscillator (clock) options.

Introduction to PIC Programming

Mid-Range Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 8: Reset, Power and Clock Options

The lessons until now have focussed on programming, but as engineers we also need to consider some of the “hardware” aspects of designing with mid-range PIC microcontrollers.

PICs require an oscillator to drive the processor clock, and although we have been using the internal RC oscillator so far, it is not always the most appropriate choice, as we will see in this lesson.

And of course PICs, like all electronic devices, need a reliable power supply. Program execution should not commence until the power supply is stable, and it may be appropriate to hold the device in a reset state if the power supply sags (a *brown-out*), to prevent unreliable operation.

In summary, this lesson covers:

- Oscillator (clock) options
- Power-on reset (POR)
- Power-up timer (PWRT)
- Brown-out detection (BOD)

Oscillator (Clock) Options

Every example in this tutorial series, until now, has used the PIC12F629’s 4 MHz internal RC oscillator as the processor clock source. It’s often a very good option – simple to use, needing no external components, using none of the PIC pins, and reasonably accurate.

However, there are situations where it is more appropriate to use some external oscillator circuitry.

Reasons to use external oscillator circuitry include:

- *Greater accuracy and stability.*
A crystal or ceramic resonator is significantly more accurate than the internal RC oscillator, with less frequency drift due to temperature and voltage variations.
- *Generating a specific frequency.*
For example, as we saw in [lesson 4](#), the signal from a 32.768 kHz crystal can be readily divided down to 1 Hz. Or, to produce accurate timing for RS-232 serial data transfers, a crystal frequency such as 1.843200 MHz can be used, since it is an exact multiple of common bit rates, such as 38400 or 9600 ($1843200 = 48 \times 38400 = 192 \times 9600$).
- *Synchronising with other components.*
Sometimes it simplifies design if a number of microcontrollers (or other chips) are clocked from a common source, so that their outputs change synchronously – although you need to be careful; clock

signals which are subject to varying delays in a circuit will not be synchronised in practice (a phenomenon known as *clock skew*), leading to unpredictable results.

Another approach is to make the PIC's clock available externally, so that other components can be synchronised with it.

- *Lower power consumption.*

At a given supply voltage, PICs draw less current when they are clocked at a lower speed. For example, the PIC12F629/675 data sheet states (parameter D015) that, with $V_{DD} = 2.0$ V, supply current is typically 340 μ A when using the internal 4MHz RC oscillator, but only 9 μ A when a 32 kHz crystal oscillator is used.

Power consumption can be minimised by running the PIC at the slowest practical clock speed and power supply voltage. And for many applications, very little speed is needed.

- *Faster operation.*

Most mid-range PICs can operate at a clock rate of up to 20 MHz, while the internal RC oscillator generally runs at only 4 or 8 MHz. If you need more speed than the internal oscillator can provide, you need to use a crystal or other external clock source.

Mid-range PICs support a number of clock, or oscillator, configurations, allowing, through appropriate oscillator selection, any of these goals to be met (but not necessarily all at once – low power consumption and high frequencies don't mix!)

The oscillator configuration is selected by the FOSC bits in the configuration word:

Bit 13	12	11	10	9	8	7	6	5	4	3	2	1	Bit 0
BG1	BG0	-	-	-	$\overline{\text{CPD}}$	$\overline{\text{CP}}$	BODEN	MCLRE	$\overline{\text{PWRTE}}$	WDTE	FOSC2	FOSC1	FOSC0

The PIC12F629 has three FOSC bits, allowing selection of one of eight oscillator configurations, as in the table below:

FOSC<2:0>	Standard MPASM symbol	Oscillator configuration
000	<code>_LP_OSC</code>	LP oscillator
001	<code>_XT_OSC</code>	XT oscillator
010	<code>_HS_OSC</code>	HS oscillator
011	<code>_EC_OSC</code>	EC oscillator
100	<code>_INTRC_OSC_NOCLKOUT</code>	Internal RC oscillator + GP4
101	<code>_INTRC_OSC_CLKOUT</code>	Internal RC oscillator + CLKOUT
110	<code>_EXTRC_OSC_NOCLKOUT</code>	External RC oscillator + GP4
111	<code>_EXTRC_OSC_CLKOUT</code>	External RC oscillator + CLKOUT

Internal RC oscillator

We have become familiar with the 4 MHz internal RC oscillator, but as you can see, it is available in two configurations.

In the first (the configuration we have been using), the internal RC oscillator provides a (nominal) 4 MHz processor clock (FOSC), which is used to drive the execution of instructions at (nominally) 1 MHz.

In the second configuration, ‘_INTRC_OSC_CLKOUT’, this instruction clock (FOSC/4) is output on the CLKOUT pin, to allow external devices to be synchronised with the PIC’s instruction clock.

Since, on the 12F629, CLKOUT shares pin 3, GP4 cannot be used for I/O in ‘_INTRC_OSC_CLKOUT’ mode.

To relate the signal we see back to the instruction clock rate, it’s useful to toggle a pin as quickly as possible, for comparison with CLKOUT, using a simple program such as:

```

;*****
; Description: Lesson 8, example 1 *
; *
; Demonstrates CLKOUT function in Internal RC oscillator mode *
; *
; Toggles a pin as quickly as possible (0.167 MHz) *
; for comparison with 1 MHz CLKOUT signal *
; *
;*****
; Pin assignments: *
; GP2 = 0.167 MHz output *
; CLKOUT = 1 MHz clock output *
; *
;*****

list p=12F629
#include <p12F629.inc>

errorlevel -302 ; no warnings about registers not in bank 0

;***** CONFIGURATION
; ext reset, no code or data protect, no brownout detect,
; no watchdog, power-up timer, 4 Mhz int clock with CLKOUT
__CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_CLKOUT

; pin assignments
constant nOUT=2 ; fast-changing (0.167 MHz) output on GP2

;***** RESET VECTOR *****
RESET CODE 0x0000 ; processor reset vector
; calibrate internal RC oscillator
call 0x03FF ; retrieve factory calibration value
banksel OSCCAL ; (stored at 0x3FF as a retlw k)
movwf OSCCAL ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start
; configure port
banksel TRISIO ; configure all pins (except GP3 and GP4/CLKOUT)
clrf TRISIO ; as outputs

;***** Main loop
movlw 1<<nOUT ; toggle output pin
banksel GPIO
loop xorwf GPIO,f ; as fast as possible
goto loop

END

```


The internal RC oscillator with CLKOUT configuration was selected by:

```

; ext reset, no code or data protect, no brownout detect,
; no watchdog, power-up timer, 4 Mhz int clock with CLKOUT
__CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_CLKOUT

```

To toggle the GP2 pin as quickly as possible, the main loop was made as tight as possible:

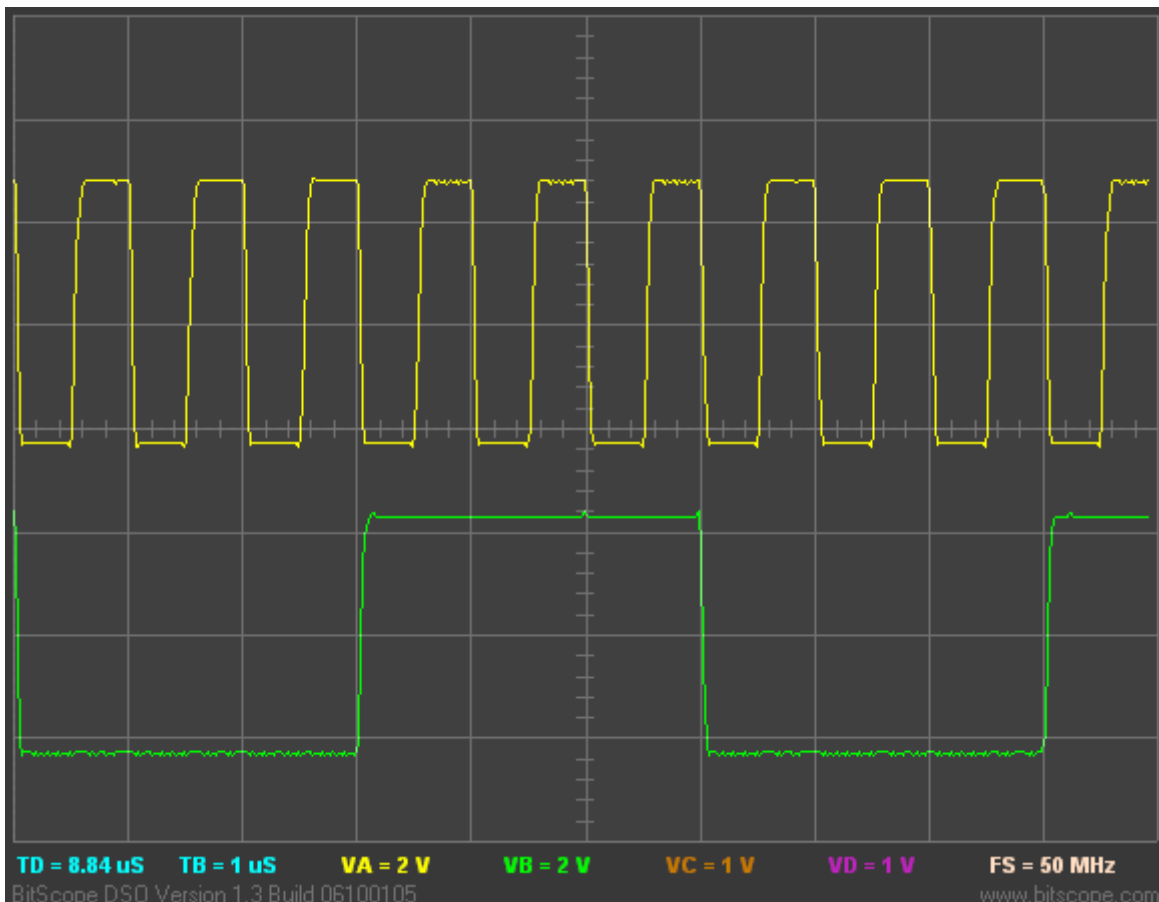
```

loop    xorwf    GPIO,f          ; as fast as possible
        goto    loop

```

With only two instructions in the loop, GP2 is toggled every three cycles (one cycle for the `xorwf` instruction and two for the `goto`), i.e. every 3 μ s, at a frequency of approx. 166 kHz.

This is apparent in the following oscilloscope plot:



The top trace is the instruction clock signal on CLKOUT, which, as you can see, has a period very close to 1 μ s, giving a frequency of 1 MHz, as expected. The bottom trace is the signal on GP2, which changes state every three instruction cycles, as expected. Also note that the transitions on GP2 are aligned with the falling edge of the instruction clock on CLKOUT.

These signals are available on pins 3 ('GP/RA/RB4') and 13 ('GP/RA/RB2') of the 16-pin header on the [Gooligum training board](#); the ground reference is pin 16 ('GND').

External clock input

An external oscillator can be used as the PIC's clock source.

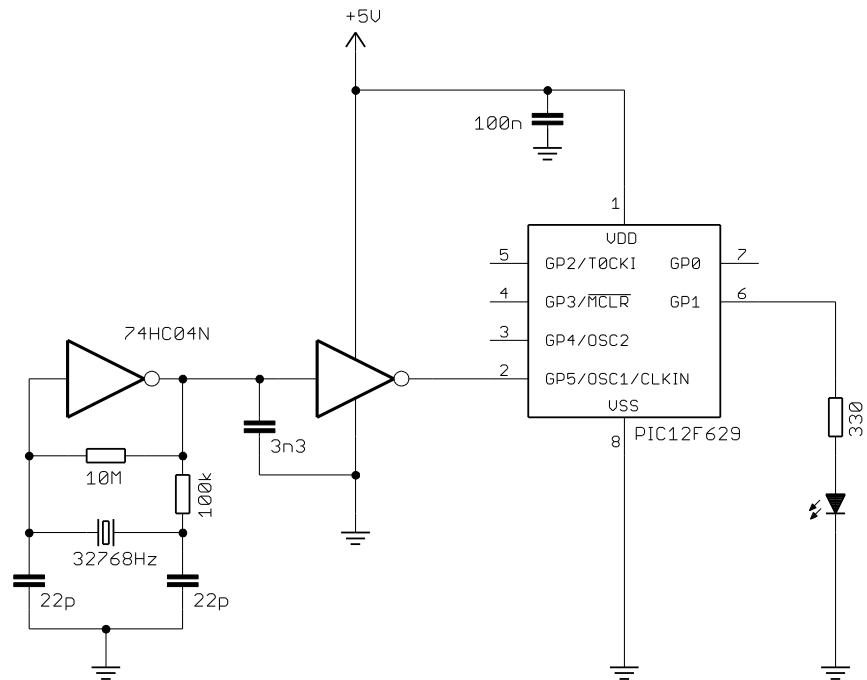
This is sometimes done so that various parts of a circuit are synchronised to the same clock signal. Or, you may choose to use an existing external clock signal simply because it is available and is more accurate and

stable than the PIC's internal RC oscillator – assuming that you can afford to lose the use of one of the PIC's pins for I/O.

[Baseline lesson 5](#) included a 32.768 kHz crystal oscillator, as shown in the circuit on the right (the reset switch and pull-up are omitted for clarity). We can use it to demonstrate how to use an external clock signal.

To use an external oscillator with the PIC12F629, the 'EC' oscillator mode should be used, with the clock signal (with a frequency of up to 20 MHz) connected to the CLKIN input: pin 2 on a PIC12F629.

To implement this circuit using the [Gooligum training board](#), place a shunt in position 4 ("EC") of jumper block JP20, connecting the 32.768 kHz signal to CLKIN, and in JP3 and JP12 to enable the external MCLR pull-up resistor (not shown here) and the LED on GP1.



Since CLKIN uses the same pin as GP5, GP5 cannot be used for I/O when the PIC is in 'EC' mode.

Note that it is also possible to use an external clock to drive CLKIN in the 'LP', 'XT' and 'HS' oscillator modes, but in those modes the OSC2 pin (pin 3 on a PIC12F629) must be left disconnected and the associated I/O pin (GP4) is not available for use – so it is much better to choose the 'EC' mode when you are using an external clock source.

To illustrate the operation of this circuit, we can modify the crystal-driven LED flasher program developed in [lesson 4](#). In that example, the external 32.768 kHz signal was used to drive the Timer0 counter.

Now, however, the 32.768 kHz signal is driving the processor clock, giving an instruction clock rate of 8192 Hz. If Timer0 is configured in timer mode with a 1:32 prescale ratio, TMR0<7> will be cycling at exactly 1 Hz (since $8192 = 32 \times 256$) – as is assumed in the example from [lesson 4](#).

Therefore, to adapt that program for this circuit, all we need to do is to change the configuration statement to:

```

; ext reset, no code or data protect, no brownout detect,
; no watchdog, power-up timer, external clock
__CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _EC_OSC

```

And also to change the initialisation code from:

```

movlw    b'11110110'    ; configure Timer0:
; --1-----    counter mode (T0CS = 1)
; ----0---    prescaler assigned to Timer0 (PSA = 0)
; -----110    prescale = 128 (PS = 110)
banksel  OPTION_REG    ; -> incr at 256 Hz with 32.768 kHz input
movwf   OPTION_REG

```

to:

```

movlw    b'11000100'    ; configure Timer0:
; --0-----            timer mode (T0CS = 0)
; ----0----            prescaler assigned to Timer0 (PSA = 0)
; -----100           prescale = 32 (PSA = 100)
banksel  OPTION_REG     ; -> incr at 256 Hz with 8192 Hz inst clock
movwf   OPTION_REG

```

Since we are no longer using the internal RC oscillator, there is no need to include the usual OSCCAL calibration code at the start of the program; this routine can safely be removed.

With these changes made, the LED on GP1 should flash at almost exactly 1 Hz – to within the accuracy of the crystal oscillator.

Complete program

Here is the program from [lesson 4](#), modified as described above:

```

;*****
;
; Description: Lesson 8, example 2
;
; Demonstrates use of external clock mode
; (using 32.768 kHz clock source)
;
; LED flashes at 1 Hz (50% duty cycle),
; with timing derived from 8192 Hz instruction clock
;
;*****
;
; Pin assignments:
; GP1 = flashing LED
; CLKIN = 32.768 kHz signal
;
;*****

list      p=12F629
#include  <p12F629.inc>

errorlevel -302          ; no "register not in bank 0" warnings

;***** CONFIGURATION
; ext reset, no code or data protect, no brownout detect,
; no watchdog, power-up timer, external clock
__CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _EC_OSC

;***** VARIABLE DEFINITIONS
UDATA_SHR
temp      res 1          ; temp register used for rotates

;***** RESET VECTOR *****
RESET    CODE    0x0000    ; processor reset vector

;***** MAIN PROGRAM *****

```

```

;***** Initialisation
start
    ; configure port
    movlw    ~(1<<GP1)          ; configure GP1 (only) as an output
    banksel TRISIO
    movwf   TRISIO

    ; configure timer
    movlw   b'11000100'        ; configure Timer0:
                                ; timer mode (T0CS = 0)
                                ; prescaler assigned to Timer0 (PSA = 0)
                                ; prescale = 32 (PS = 100)
    banksel OPTION_REG        ; -> increment at 256 Hz with 8192 Hz inst
clock
    movwf   OPTION_REG

;***** Main loop
main_loop
    ; TMR0<7> cycles at 1Hz, so continually copy to LED (GP1)
    banksel TMR0
    rlf     TMR0,w             ; copy TMR0<7> to C
    clrf   temp
    rlf    temp,f             ; rotate C into temp
    rlf    temp,w             ; rotate once more into W (-> W<1> = TMR0<7>)
    movwf  GPIO               ; update GPIO with result (-> GP1 = TMR0<7>)

    ; repeat forever
    goto   main_loop

END

```

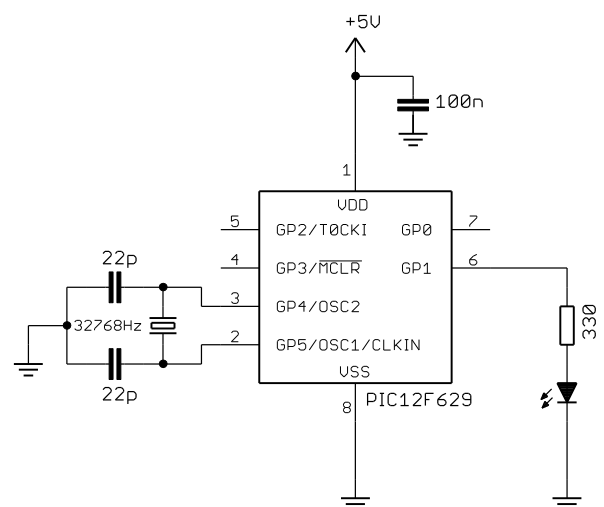
Crystals and ceramic resonators

Generally, there is no need to build your own crystal oscillator; PICs include an oscillator circuit designed to drive crystals directly.

A parallel (not serial) cut crystal, or a ceramic resonator, is placed between the OSC1 and OSC2 pins, which are grounded via loading capacitors, as shown in the circuit diagram (with the reset switch and pull-up omitted for clarity) on the right.

You should consult the crystal or resonator manufacturer's data when selecting load capacitors; those shown here are appropriate for a crystal designed for a load capacitance of 12.5 pF.

For some crystals it may be necessary to reduce the drive current by placing a resistor between OSC2 and the crystal, but in most cases it is not needed, and the circuit shown here can be used.



If you are using the [Gooligum training board](#), place shunts in position 2 (“32kHz”) of JP20¹ and position 2 of JP21 (“32kHz”), connecting the 32.768 kHz crystal between OSC1 and OSC2, and close JP3 and JP12 to enable the external $\overline{\text{MCLR}}$ pull-up resistor (not shown here) and the LED on GP1.

The PIC12F629 offers three crystal oscillator modes: ‘XT’, ‘LP’ and ‘HS’. They differ in the gain and frequency response of the drive circuitry.

‘XT’ (“crystal”) is the mode used most commonly for crystals or ceramic resonators operating between 100 kHz and 4 MHz.

‘HS’ (“high speed”) mode provides higher gain and is typically used for crystals or ceramic resonators operating above 4 MHz, up to a maximum frequency of 20 MHz. Because of the higher drive level, a series resistor is more likely to be necessary in ‘HS’ oscillator mode.

Lower frequencies generally require lower gain. The ‘LP’ (“low power”) mode uses less power and is designed to drive common 32.786 kHz “watch” crystals, as used in the external clock circuit above, although it can also be used with other low-frequency crystals or resonators.

The circuit as shown here can be used to operate the PIC12F629 at 32.768 kHz, giving low power consumption and an 8192 Hz instruction clock rate, which, as in the external clock example, is easily divided to create an accurate 1 Hz signal.

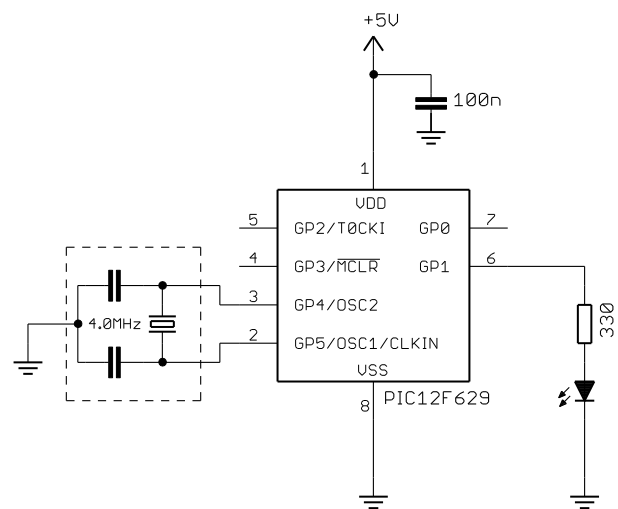
To flash the LED at 1 Hz, the program is exactly the same as for the external clock example above, except that the configuration statement must instead include the `_LP_OSC` option:

```
__CONFIG MCLRRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _LP_OSC
```

A convenient option, when you want greater accuracy and stability than the internal RC oscillator can provide, but do not need as much as that offered by a crystal, is to use a ceramic resonator.

These are available in convenient 3-terminal packages which include appropriate loading capacitors, as shown in the circuit diagram (with the reset switch and pull-up omitted for clarity) on the right. The resonator package incorporates the components within the dashed lines.

If you have the [Gooligum training board](#), move the shunts to position 3 (“4MHz”) of JP20 and position 1 of JP21 (“4MHz”), connecting the 4.0 MHz resonator between OSC1 and OSC2, and leave JP3 and JP12 closed to enable the external $\overline{\text{MCLR}}$ pull-up resistor (not shown here) and the LED on GP1.



¹ You will find, with the Gooligum training board that the LED in this 32.768 kHz crystal example will flash, even with no shunt installed in JP20! This is because, when configured in `_LP_OSC` mode, the OSC1 input is very sensitive, and picks up crosstalk from the external 32.768 kHz signal on the board. If you want to prevent this effect, you can dampen the external 32.768 kHz signal by loading it with a 100 Ω resistor, placed between pin 1 of the 16-pin expansion header and ground, via the breadboard. The external clock example will still work with this resistor in place, and this 32.768 kHz crystal example will only work with shunts in the “32kHz” positions of JP20 and JP21 – as we’d expect.

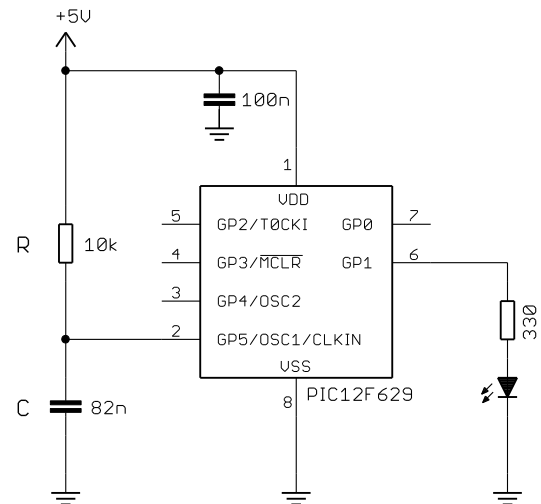
To test this circuit, you can change the ‘`_INTRC_OSC_NOCLKOUT`’ configuration option to ‘`_XT_OSC`’ in one of the LED flashing examples from earlier lessons, since they are already written for use with a 4 MHz clock (we’re simply using an external resonator instead of the internal RC oscillator).

A good choice is the “flash an LED at exactly 1 Hz” program developed in [lesson 6](#), since it will generate an output of exactly 1 Hz, given a processor clock of exactly 4 MHz, and so should benefit from this more accurate clock source.

External RC oscillator

Finally, a low-cost, low-power option: mid-range PICs can use an oscillator based on an external resistor and capacitor, as shown (with the reset switch and pull-up omitted for clarity) on the right.

To implement this circuit using the [Gooligum training board](#), move the shunt to position 1 (“RC”) of JP20, connecting the 10 kΩ resistor and 82 nF capacitor to OSC1. Remove the shunt from JP21 and leave JP3 and JP12 closed, enabling the external `MCLR` pull-up resistor (not shown here) and the LED on GP1.



External RC oscillators were once the only alternative to a crystal or resonator, but the availability of internal RC oscillators in modern PICs has meant that they are much less commonly used these days.

However, external RC oscillators, with appropriate values of R and C, can still be useful when a very low clock rate is acceptable – drawing significantly less power than when the internal 4 MHz RC oscillator is used.

Running the PIC slowly can also simplify some programming tasks.

The external RC oscillator is a *relaxation* type.

The capacitor is charged through the resistor, the voltage v at the OSC1 pin rising with time t according to the formula:

$$v = V_{DD} \left(1 - e^{-t/RC} \right)$$

The voltage increases until it reaches a threshold, typically $0.75 \times V_{DD}$. A transistor is then turned on, which quickly discharges the capacitor until the voltage falls to approx. $0.25 \times V_{DD}$. The capacitor then begins charging through the resistor again, and the cycle repeats.

In theory, assuming upper and lower thresholds of $0.75 \times V_{DD}$ and $0.25 \times V_{DD}$, the period of oscillation is equal to $1.1 \times RC$ (in seconds, with R in Ohms and C in Farads).

In practice, the capacitor discharge is not instantaneous (and of course it can never be), so the period is a little longer than this. Microchip does not commit to a specific formula for the frequency (or period) of the external RC oscillator, only stating that it is a function of VDD, R, C and temperature, and in some documents providing some reference charts. But for rough design guidance, you can assume the period of oscillation is approximately $1.2 \times RC$.

Microchip recommends keeping R between 5 kΩ and 100 kΩ, and C above 20 pF.

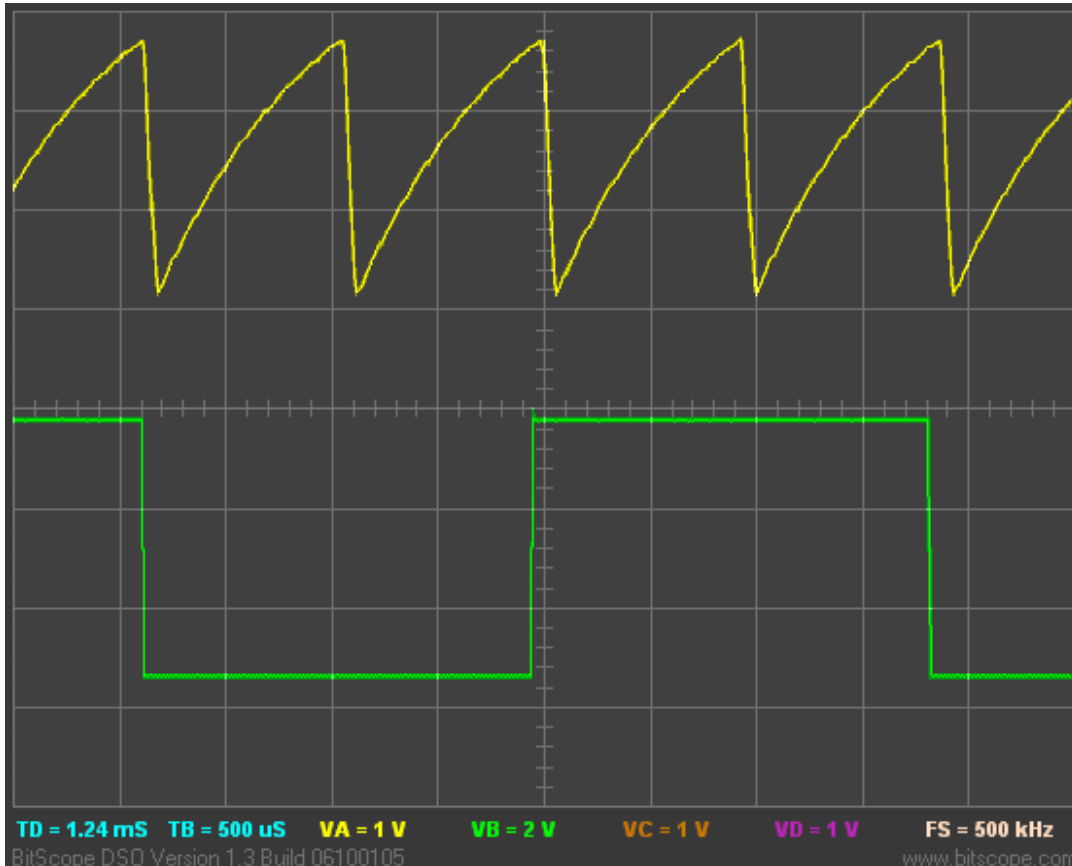
In this circuit above, R = 10 kΩ and C = 82 nF.

Those values will give a period of approximately:

$$1.2 \times 10 \times 10^3 \times 82 \times 10^{-9} \text{ s} = 984 \mu\text{s}$$

Hence, we can expect to generate a clock frequency of around 1 kHz.

This circuit was tested, using the component values shown, giving the following oscilloscope traces:



The top trace was recorded at the OSC1 pin, and shows the expected RC charge/discharge cycles.

The bottom trace shows the instruction clock output at the CLKOUT pin; as expected, it is one quarter of the frequency of the clock input at OSC1.

In practice, the measured frequency was 1080 Hz; reasonably close, but the lesson should be clear: don't use an external RC oscillator if you want high accuracy or good stability.

Only use an external RC oscillator if the exact clock rate is unimportant.

So, given a roughly 1 kHz clock, what can we do with it? Flash an LED, of course!

Using a similar approach to before, we can use the instruction clock (approx. 256 Hz) to increment Timer0. In fact, with a prescale ratio of 1:256, TMR0 will increment at approx. 1 Hz.

TMR0<0> would then cycle at 0.5 Hz, TMR0<1> at 0.25 Hz, etc.

Now consider what happens when the prescale ratio is set to 1:64. TMR0 will increment at 4 Hz, TMR0<0> will cycle at 2 Hz, and TMR0<1> will cycle at 1 Hz, etc.

And that suggests a very simple way to make the LED on GP1 flash at 1 Hz: if we continually copy TMR0 to GPIO, each bit of GPIO will reflect each corresponding bit of TMR0.

In particular, GPIO<1> will always be set to the same value as TMR0<1>. Since TMR0<1> is cycling at 1 Hz, GPIO<1> (and hence GP1) will also cycle at 1 Hz.

Complete program

The following program implements the approach described above. Note that the external RC oscillator is selected by using the option `_EXTRC_OSC_CLKOUT` in the configuration statement.

```

;*****
;
; Description: Lesson 8, example 5
;
; Demonstrates use of external RC oscillator (~1 kHz)
;
; LED on GP1 flashes at approx 1 Hz (50% duty cycle),
; with timing derived from instruction clock
;
;*****
;
; Pin assignments:
; GP1 = flashing LED
; OSC1 = R (10k) / C (82n)
;
;*****

list p=12F629
#include <p12F629.inc>

errorlevel -302 ; no "register not in bank 0" warnings

;***** CONFIGURATION
; ext reset, no code or data protect, no brownout detect,
; no watchdog, power-up timer, ext RC oscillator (~1kHz) +
clkout
__CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _EXTRC_OSC_CLKOUT

;***** RESET VECTOR *****
RESET CODE 0x0000 ; processor reset vector

;***** MAIN PROGRAM *****

;***** Initialisation
start
; configure port
movlw ~(1<<GP1) ; configure GP1 (only) as an output
banksel TRISIO
movwf TRISIO

; configure timer
movlw b'11000101' ; configure Timer0:
; --0----- timer mode (T0CS = 0)
; ----0--- prescaler assigned to Timer0 (PSA = 0)
; -----101 prescale = 64 (PS = 101)
banksel OPTION_REG ; -> increment at 4 Hz with 1 kHz clock
movwf OPTION_REG

```



```

;***** Main loop
main_loop
    ; TMR0<1> cycles at 1Hz, so continually copy to LED (GP1)
    banksel TMR0
    movf    TMR0,w          ; copy TMR0 to GPIO
    movwf  GPIO

    ; repeat forever
    goto   main_loop

END

```

The “main loop” is only three instructions long – by far the shortest “flash an LED” program we have done so far, illustrating how a slow clock rate can sometimes simplify some programming problems. On the other hand, it is also the least accurate of the “flash an LED” programs, being only approximately 1 Hz. But for many applications, the exact speed doesn’t matter; it only matters that the LED visibly flashes, not how fast.

Power-On Reset

When we apply power to a PIC, we expect it to start executing whatever program is loaded into it, and to do so reliably, every time.

Unfortunately, it may not be that simple. “Apply power” is not the same as “instant on”. Real power supplies have a source impedance, and the circuits attached to them have some capacitance, so it takes some time for the power supply to ramp up to its final voltage. And while doing so, devices in the circuit will begin drawing current unevenly as they come to life, meaning that the power supply may initially be unsteady, taking some time to settle, as the circuit reaches equilibrium.

That can be a problem for devices, such as PICs, which have a minimum operating voltage. For example, the PIC12F629 requires a power supply of at least 2.0 V when running at a clock rate of 4 MHz or less. But at least 4.5 V is required for operation at frequencies above 10 MHz.

Below these voltages, the device may operate, but not reliably. Some parts of it may operate, while others fail. For example, the analog-to-digital converter (ADC) on the PIC12F675 requires at least 2.5 V, while other parts of the device (which is essentially the same as a 12F629) may operate down to 2.0 V.

And if the power supply is unstable while it ramps up, the PIC may drop in and out of operation while this minimum operating voltage region is crossed.

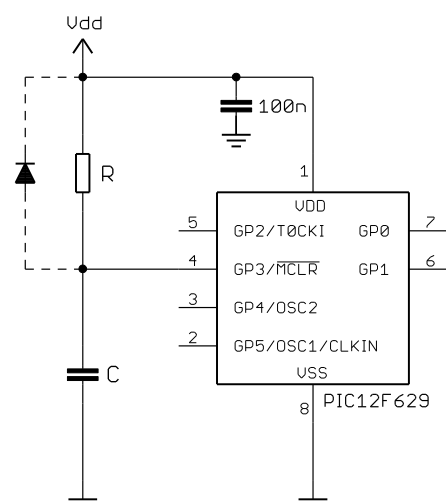
For reliable start-up, it is necessary to hold the PIC in a reset condition until the power supply has reached a high enough, and stable, voltage.

This was traditionally done by a simple RC circuit connected to the external $\overline{\text{MCLR}}$ pin, as shown on the right.

The capacitor is initially discharged, so $\overline{\text{MCLR}}$ is initially low, and remains low as the capacitor charges through the resistor. Values of R and C are chosen so that $\overline{\text{MCLR}}$ goes high after enough time has elapsed for the power supply to reach an adequate voltage and settle.

Microchip recommends that R be at least 1 k Ω for adequate ESD (electrostatic discharge) protection, and below 40 k Ω to avoid too large a voltage drop across the resistor.

The diode across the resistor is optional, being used to help discharge the capacitor more quickly when power is removed.



However, modern mid-range PICs have less need for these external reset components, because they include a *power-up timer* (PWRT), which, if enabled, holds the device in reset for a nominal 72 ms from the initial *power-on reset* (POR) which occurs when power-on is detected.

To enable the power-up timer, clear the $\overline{\text{PWRTE}}$ bit in the processor configuration word:

Bit 13	12	11	10	9	8	7	6	5	4	3	2	1	Bit 0
BG1	BG0	-	-	-	$\overline{\text{CPD}}$	$\overline{\text{CP}}$	BODEN	MCLR $\overline{\text{E}}$	$\overline{\text{PWRTE}}$	WDTE	FOSC2	FOSC1	FOSC0

Setting $\overline{\text{PWRTE}}$ to '1' *disables* the power-on timer.

To enable the power-up timer, use the symbol ' $\overline{\text{PWRTE_ON}}$ ' in the `__CONFIG` directive.

To disable it, use ' $\overline{\text{PWRTE_OFF}}$ ' instead.

So why would you ever want to disable the power-up timer?

Note that the PWRT begins to operate from the moment when the PIC detects a power-on condition, and for that to happen (for the PIC12F629), VDD has to rise from VSS at a minimum rate of 0.05 V/ms. If these conditions are not met, the power-on condition may not be detected; the power-on reset will not occur, and the PIC will not start properly. In this case, you would have to use an external circuit to hold the PIC in reset when power is applied.

Or, your power supply may take more than 72 ms to settle. And note that this is a nominal value – the actual PWRT delay on a PIC12F629 may be as short as 28 ms. If the power supply has not stabilised in this time, an external reset circuit should be used to hold the device in reset for longer.

Or, you may be using an external supervisory circuit, such as one of Microchip's MCP10X devices.

If, for any of these reasons, you are using an external circuit which holds the PIC in reset during power-up, it may appropriate to disable the internal power-up timer, so that there is only one source of power-up delay.

But most of the time, unless your circuit is operating in difficult power supply conditions, you can enable the power-up timer (as we have done so far) and, if you are using an external reset, use a 10 k Ω resistor between $\overline{\text{MCLR}}$ and VDD.

If you are using the LP, XT or HS clock mode (which implies that you're probably using a crystal or resonator driven by the PIC's on-board oscillator circuitry), the *oscillator start-up timer* (OST) is invoked to give the crystal or resonator time to settle, after the PWRT delay completes.

The OST counts pulses on the OSC1 pin, and holds the device in reset for 1024 oscillator cycles. Hence, the OST delay depends on the clock speed. With a 4 MHz resonator, the OST delay is only 256 μs , while the delay with a 32.768 kHz crystal is 31 ms. Note that these delay times are nominal; after all, the reason the OST is invoked is that it takes a while for a crystal or resonator to begin stable oscillation, so in practice the delay times will probably be longer than this.

The OST is also used when the PIC wakes from sleep in LP, XT or HS clock mode, for the same reason – the oscillator is disabled while the device is in sleep mode, and takes a while to start and become stable.

Note that the OST is invoked whether or not PWRT is enabled. The only way to avoid the oscillator start-up delay is to use one of the EC, internal RC or external RC oscillator modes.

For fastest processor start-up at power-on, disable the power-up timer and use an external clock, avoiding both the PWRT and OST delays – and hope that you have a very fast-starting and stable power supply! But it's generally best to simply accept that your program won't start running for up to 100 ms after you turn the power on...

Brown-out Detect

We've seen that the PIC should be held in reset during power-up, to avoid instability while the power supply is ramping up through the device's minimum operating voltage.

Similarly, the PIC's operation can become unreliable if the power supply voltage falls too far during normal operation – a condition known as a *brown-out*. It can happen when the load on the power supply varies; battery-powered systems can be susceptible to this, particularly when the batteries are running down, as well as industrial or automotive settings where large loads are switched in (think of the headlights of your car dimming when you use the starter motor).

In general, it is preferable to stop program execution as long as the brown-out situation persists, instead of risking unreliable operation; it's better to be able to recover cleanly after the brown-out, instead of not knowing what your program might do.

Most mid-range PICs provide a *brown-out detect* (BOD, also called *brown-out reset*, or BOR) facility, which, if enabled, will reset the device if the supply voltage falls below the brown-out detect voltage (between 2.025 V and 2.175 V on the PIC12F629), and hold it in reset until the voltage rises again. If the power-up timer is enabled (recommended if you are using BOD), the device will remain in reset for a further 72 ms after the brown-out condition clears – and if another brown-out occurs during this PWRT delay, it will be detected and the process will repeat.

To enable brown-out detect on the PIC12F629, set the **BODEN** bit in the processor configuration word:

Bit 13	12	11	10	9	8	7	6	5	4	3	2	1	Bit 0
BG1	BG0	-	-	-	$\overline{\text{CPD}}$	$\overline{\text{CP}}$	BODEN	MCLRE	$\overline{\text{PWRTE}}$	WDTE	FOSC2	FOSC1	FOSC0

Setting **BODEN** to '1' enables brown-out detection.

To enable BOD, use the symbol '`_BODEN_ON`' in the `__CONFIG` directive.

To disable it, use '`_BODEN_OFF`' instead.

Detecting a brown-out reset

If a brown-out occurs, resetting the PIC and hence restarting your program, you may want your application to react to this, behaving differently to a power-on, watchdog timer, or other reset. In particular, if your program has restarted because of a brown-out, you may want it to try to continue doing whatever it was doing before the brown-out, instead of running through the full initialisation routine. Or you may wish to initialise external devices, which may also have been affected by the brown-out, and perhaps run through some system checks to ensure that everything is now ok. Or you might want to simply log the fact that a brown-out has occurred.

Fortunately, mid-range PICs provide flags which allow us to detect and respond differently to both power-on and brown-out resets.

In the 12F629, these flags are contained in the power control register, **PCON**:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PCON	-	-	-	-	-	-	$\overline{\text{POR}}$	$\overline{\text{BOD}}$

The $\overline{\text{POR}}$ (power-on reset status) flag is cleared when a power-on reset occurs, and is set if a brown-out reset occurs. It is unaffected by all other resets.

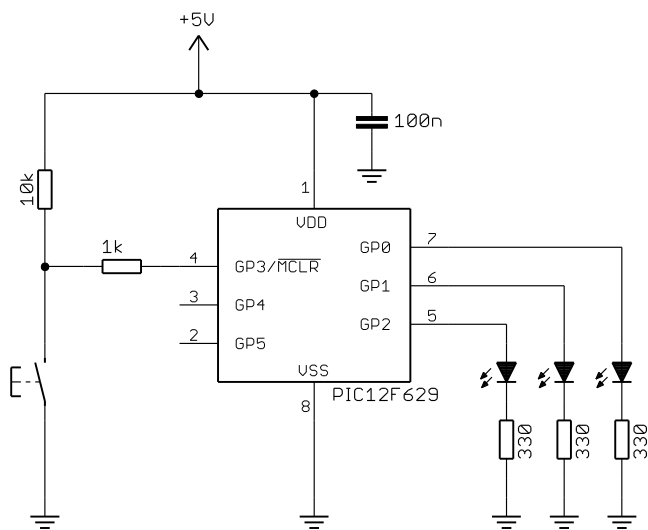
This means that, to use this flag to differentiate power-on from other resets, you must set $\overline{\text{POR}}$ to '1' whenever a power-on reset occurs. Since all the other types of reset either set this bit or leave it unchanged, it will then only ever be '0' when a power-on reset has occurred.

Similarly, the $\overline{\text{BOD}}$ (brown-out detect status) flag is cleared when a brown-out reset occurs, and is unaffected by all other resets.

To use this flag to differentiate brown-out from other resets, you must set $\overline{\text{BOD}}$ to '1' following power-on. Since all the other resets leave this bit unchanged, it will only ever be '0' when a brown-out has occurred.

Since $\overline{\text{BOD}}$ is unaffected by a power-on reset, its value is unknown when the device is first powered on. Therefore, the first flag you should test is $\overline{\text{POR}}$. If it is clear, you can be sure that a power-on reset has occurred, and you can then set both $\overline{\text{POR}}$ and $\overline{\text{BOD}}$, ready for testing after subsequent resets.

An example may help to clarify this.



We'll use the circuit shown on the left, which you can implement with the [Gooligum training board](#) by closing jumpers JP3, JP11, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP0, GP1 and GP2.

If you are using Microchip's Low Pin Count Demo Board, you can connect LEDs to GP0, GP1 and GP2, by making connections on the 14-pin header: 'RA0' to 'RC0', 'RA1' to 'RC1' and 'RA2' to 'RC2'.

The program will simply light the LED on GP0, regardless of why the PIC had been reset (or powered on).

In addition, the LED on GP1 will be lit on power-on (and not for any other reset), and the LED on GP2 will indicate that a brown-out has occurred.

The pushbutton will be used to generate an external $\overline{\text{MCLR}}$ reset. When this happens, only the LED on GP0 should light, because the reset is caused by neither power-on nor brown-out.

Brownout detection has to be enabled in the device configuration:

```

; ext reset, no code or data protect, brownout detect,
; no watchdog, power-up timer, 4 Mhz int clock
__CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_ON & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

```

After the usual initialisation code, the first task is to test the $\overline{\text{POR}}$ flag to see if a power-on reset has occurred:

```

; check for POR or BOD reset
banksel PCON
btfsc PCON,NOT_POR ; if power-on reset (NOT_POR = 0),
goto chk_bod

```

If this is a power-on reset, we should set the $\overline{\text{POR}}$ and $\overline{\text{BOD}}$ flags, to set them up for any subsequent resets (as discussed above), and light the POR LED:

```
bsf    PCON,NOT_POR    ; set POR and BOD flags for next reset
bsf    PCON,NOT_BOD
bsf    sGPIO,nP_LED    ; enable POR LED (shadow)
```

Note the use of a shadow copy of GPIO here. Since it is held in shared (unbanked) memory, updating the shadow register avoids the extra `banksel` directives we'd need if we were instead writing directly to GPIO. It also has the advantage of avoiding potential read-modify-write issues, as discussed [before](#).

Now we can reliably test for a brown-out reset:

```
chk_bod btfscc PCON,NOT_BOD    ; if brown-out detect (NOT_BOD = 0)
        goto    main
```

and, if one has occurred, set the $\overline{\text{BOD}}$ flag for next time, and light the BOD LED:

```
bsf    PCON,NOT_BOD    ; set BOD flag for next reset
bsf    sGPIO,nB_LED    ; enable BOD LED (shadow)
```

Note that, if a power-on reset had occurred, this brown-out detect code will never be executed, because the earlier code sets the $\overline{\text{BOD}}$ flag, whenever a power-on reset is detected.

Regardless of the reason for the reset, we light the "on" LED:

```
main
    ; enable "on" indicator LED
    bsf    sGPIO,nO_LED    ; (using shadow register)

    ; light enabled LEDs
    movf   sGPIO,w          ; copy shadow GPIO to port
    banksel GPIO
    movwf  GPIO
```

If the pushbutton is pressed, generating a $\overline{\text{MCLR}}$ reset, only this "on" LED will be lit.

Finally, we simply wait until the next reset:

```
    ; wait forever
    goto  $
```

Complete program

Here is how these pieces fit together:

```

;*****
;
; Description:    Lesson 8, example 6
;
; Demonstrates use of brown-out detect
; and differentiation between POR, BOD and MCLR resets
;
; Turns on POR LED only if power-on reset is detected
; Turns on BOD LED only if brown-out detect reset is detected
; Turns on indicator LED in all cases
; (no POR or BOD implies MCLR, as no other reset sources are active)

```

```

;*****
;
; Pin assignments:
; GP0 = "on" indicator LED (always turned on)
; GP1 = POR LED (indicates power-on reset)
; GP2 = BOD LED (indicates brown-out detected)
;
;*****

list      p=12F629
#include   <p12F629.inc>

errorlevel -302          ; no warnings about registers not in bank 0

;***** CONFIGURATION
                ; ext reset, no code or data protect, brownout detect,
                ; no watchdog, power-up timer, 4 Mhz int clock
__CONFIG      _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_ON & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

; pin assignments
constant      nO_LED=0          ; on indicator LED on GP0 (always on)
constant      nP_LED=1          ; POR LED on GP1 to indicate power-on reset
constant      nB_LED=2          ; BOD LED on GP2 to indicate brown-out

;***** VARIABLE DEFINITIONS
UDATA_SHR
sGPIO        res 1              ; shadow copy of GPIO

;***** RESET VECTOR *****
RESET        CODE    0x0000      ; processor reset vector
                ; calibrate internal RC oscillator
                call   0x03FF      ; retrieve factory calibration value
                banksel OSCCAL      ; (stored at 0x3FF as a retlw k)
                movwf  OSCCAL      ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start
                ; configure port
                banksel GPIO          ; start with all LEDs off
                clrf   GPIO
                clrf   sGPIO          ; update shadow
                movlw  ~(1<<nO_LED|1<<nP_LED|1<<nB_LED) ; configure LED pins as
outputs
                banksel TRISIO
                movwf  TRISIO

                ; check for POR or BOD reset
                banksel PCON
                btfsc  PCON,NOT_POR      ; if power-on reset (NOT_POR = 0),
                goto  chk_bod
                bsf   PCON,NOT_POR      ; set POR and BOD flags for next reset
                bsf   PCON,NOT_BOD
                bsf   sGPIO,nP_LED      ; enable POR LED (shadow)
chk_bod btfsc  PCON,NOT_BOD      ; if brown-out detect (NOT_BOD = 0)

```

```

        goto    main
        bsf     PCON,NOT_BOD      ; set BOD flag for next reset
        bsf     sGPIO,nB_LED     ; enable BOD LED (shadow)

;***** Main code
main
        ; enable "on" indicator LED
        bsf     sGPIO,nO_LED     ; (using shadow register)

        ; light enabled LEDs
        movf    sGPIO,w          ; copy shadow GPIO to port
        banksel GPIO
        movwf   GPIO

        ; wait forever
        goto    $

        END

```

To test this program, you will need a variable power supply.

If you have the [Gooligum training board](#), you can connect your power supply to Vdd and ground via pins 15 ('+V') and 16 ('GND') on the 16-pin expansion header.

You should find that if you set the supply to say 4 V and apply power, the POR LED (GP1) should light, along with the "on" LED (GP0)

If you then simulate a brown-out, by lowering the voltage until both LEDs turn off (at around 2 V; by this time they will be very dim, since the forward voltage of most normal-brightness LEDs is around 2 V), without taking the voltage all the way to zero, and then raise the voltage again, the BOD LED (GP2) should light, indicating that the brown-out was detected. The "on" LED should light, as always, but not POR, because this was a brown-out, not a power-on reset..

If you then turn off the power supply, and turn it back on again, the POR LED should light again, and not BOD, because this was a normal power-on, not a brown-out.

Finally, if you press the pushbutton, generating a $\overline{\text{MCLR}}$ reset, while either the POR or BOD LED is lit, all the LEDs will go out while the button is pressed, and then only the "on" LED will come on, indicating that this reset was neither a power-on nor a brown-out.

If you are able to raise the power supply voltage very slowly from zero, you may be able to get the BOD LED to light, instead of POR, if the voltage rise is too slow to trigger a power-on reset.

Conclusion

In this lesson we've seen that mid-range PICs can be clocked in a number of ways, that there are alternatives to the internal RC oscillator, such as an external clock, or the PIC's own oscillator circuitry driving a crystal, resonator, or a simple RC timing circuit – and we discussed some of the reasons for using those alternatives. We've also seen how the power-on reset brown-out detect features serve to simplify our circuits and make our designs more robust.

The [next lesson](#) will focus on comparators – the single comparator in the PIC12F629, and then in the [following lesson](#) we will introduce the 14-pin PIC16F684, which includes a dual comparator module.

Introduction to PIC Programming

Programming Mid-Range PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 1: Basic Digital I/O

The [Baseline PIC C Programming](#) tutorial series demonstrated how to program baseline PICs (such as the 12F509 and 16F506) in C, to perform the tasks covered in the [Baseline PIC Assembler](#) lessons: from flashing LEDs and reading switches, through to implementing a simple light meter with smoothed output on a multiplexed 7-segment LED display. The baseline C programming series used the “free” CCS PCB compiler bundled with MPLAB¹, and Microchip’s XC8 compiler (running in “Free mode”). We saw in that series that, although these C compilers were perfectly adequate for the simplest tasks, they faltered when it came to the more complex applications involving arrays, reflecting the difficulty of implementing a C compiler for the baseline PIC architecture.

This tutorial series revisits this material, along with other topics covered in the [Mid-Range PIC Assembler](#) lessons, using mid-range devices such as the 12F629 and 16F684. It will become apparent that the mid-range architecture is much more suitable for C programming than the baseline architecture, especially for applications which need to access more than one bank of data memory. But we will also see that, although it is often easier to program in C, in the sense that programs are shorter and more easily expressed, assembler remains the most effective way to make the most of the limited resources on these small devices, even for mid-range PICs. Nevertheless, we will see that C is a very practical alternative for most applications.

Microchip’s XC8 compiler supports all mid-range PICs, and can be operated in “Free mode” (with all optimisations disabled) for free – making it a good choice for use in these lessons.

This lesson covers basic digital I/O: flashing LEDs, responding to and debouncing switches, as covered in lessons [1](#) to [3](#) of the mid-range assembler tutorial series. You may need to refer to those lessons while working through this one.

In summary, this lesson covers:

- Introduction to the Microchip XC8 compiler
- Digital input and output
- Programmed delays
- Switch debouncing
- Using internal (weak) pull-ups

These tutorials assume a working knowledge of the C language; they **do not** attempt to teach C.

¹ CCS PCB is bundled with MPLAB 8 only, and only supports baseline PICs, so cannot be used with mid-range PICs.

Introducing the XC8 Compiler

Up until version 8.10, MPLAB was bundled with HI-TECH's "PICC-Lite" compiler, which supported all the baseline (12-bit) PICs available at that time, including those used in this tutorial series, with no restrictions. It also supports a small number of the mid-range (14-bit) PICs – although, for most of the mid-range devices it supported, PICC-Lite limited the amount of data and program memory that could be used, to provide an incentive to buy the full compiler. Microchip have since acquired HI-TECH Software, and no longer supply or support PICC-Lite. As such, PICC-Lite will not be covered in these tutorials.

XC8 supports the whole 8-bit PIC10/12/16/18 series in a single edition, with different licence keys unlocking different levels of code optimisation – "Free" (free, but no optimisation), "Standard" and "PRO" (most expensive and highest optimisation).

Microchip XC compilers are also available for the PIC24, dsPIC and PIC32 families.

XC8's "Free mode" supports all 8-bit (including baseline and mid-range) PICs, with no memory restrictions. However, in this mode, most compiler optimisation is turned off, making the generated code around twice the size of that generated by PICC-Lite.

This gives those developing for baseline and mid-range PICs easy access to a free compiler supporting a much wider range of devices than PICC-Lite does, without memory usage restrictions, albeit at the cost of much larger generated code. And XC8 will continue to be maintained, supporting new baseline and mid-range devices over time.

But if you are using Windows and developing code for a supported mid-range PIC, it is quite valid to continue to use PICC-Lite (if you are able to locate a copy – by downloading MPLAB 8.10 from the archives on www.microchip.com, for example), since it will generate much more efficient code. It can be installed alongside XC8. But to repeat – PICC-Lite won't be described in these lessons.

Regardless of which version of MPLAB you are using, the XC8 installer (for Windows, Linux or Mac) has to be downloaded separately from www.microchip.com.

When you run the XC8 installer, you will be asked to enter a license activation key. Unless you have purchased the commercial version, you should leave this blank. You can then choose whether to run the compiler in "Free mode", or activate an evaluation license. We'll be using "Free mode" in these lessons, but it's ok to use the evaluation license (for 60 days) if you choose to.

See [baseline assembler lesson 1](#) for more detail on installing and using MPLAB 8 or MPLAB X.

Data Types

One of the problems with implementing ANSI-standard C on microcontrollers is that there is often a need to work with individual bits, while the smallest data-type included in the ANSI standard is 'char', which is normally considered to be a single byte, or 8 bits. Another problem is the length of a standard integer ('int') is not defined, being implementation-dependent. Whether an 'int' is 16 or 32 bits is an issue on larger systems, but it makes a much more significant difference to code portability on microcontrollers. Similarly, the sizes of 'float', 'double', and the effect of the modifiers 'short' and 'long' is not defined by the standard.

So various compilers use different sizes for the "standard" data types, and for microcontroller implementations it is common to add a single-bit type as well – generally specific to that compiler.

Here are the data types and sizes supported by XC8 and, for comparison, the size of the same data types in CCS PCB:

Type	XC8	CCS PCB
bit	1	-
char	8	8
short	16	1
int	16	8
short long	24	-
long	32	16
float	24 or 32	32
double	24 or 32	-

You'll see that very few of these line up; the only point of agreement is that 'char' is 8 bits!

XC8 defines a single 'bit' type, unique to XC8.

The "standard" 'int' type is 16 bits in XC8, but 8 bits in CCS PCB.

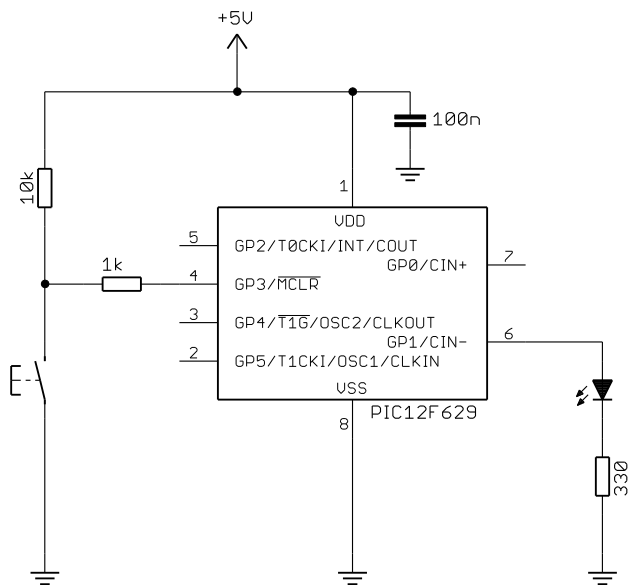
But by far the greatest difference is in the definition of 'short': in XC8, it is a synonym for 'int', with 'short', 'int' and 'short int' all being 16-bit quantities, whereas in CCS PCB, 'short' is a single-bit type.

Finally, note that 'double' floating-point variables in XC8 can be either 24 or 32 bits; this is set by a compiler option. 32 bits may be a higher level of precision than is needed in most applications for small applications, so XC8's ability to work with 24-bit floating point numbers can be useful.

To make it easier to create portable code, XC8 provides the 'stdint.h' header file, which defines the C99 standard types such as 'uint8_t' and 'int16_t'.

Example 1: Turning on an LED

We saw in [mid-range assembler lesson 1](#) how to turn on a single LED, and leave it on; the (very simple) circuit is shown below:



The LED, with a current-limiting resistor, is connected to the GP1 pin of a PIC12F629.

The pushbutton acts as a reset switch, and the 10 kΩ pull-up resistor holds MCLR high while the switch is open².

If you are using the [Gooligum training board](#), plug your PIC12F629 into the top section of the 14-pin IC socket – the section marked '12F'³. Close jumpers JP3 and JP12 to bring the 10 kΩ resistor into the circuit and to connect the LED to GP1, and ensure that every other jumper is disconnected.

If you have the Microchip Low Pin Count Demo Board, refer back to [baseline assembler lesson 1](#) to see how to build this circuit.

² This external pull-up resistor wasn't needed in the baseline PIC examples, because the baseline PICs, and indeed most mid-range PICs, include an internal weak pull-up (see example 6, later in this lesson) on MCLR which is automatically enabled whenever the device is configured for external reset.

³ Note that, although the PIC12F629 comes in an 8-pin package, **it will not work** in the 8-pin '10F' socket. You must install it in the '12F' section of the 14-pin socket.

To turn on the LED, we loaded the TRISIO register with 111101b, so that only GP1 is set as an output, and then set bit 1 of GPIO, setting GP1 high, turning the LED on.

At the start of the program, the PIC's configuration was set, and the OSCCAL register was loaded with the factory calibration value.

Finally, the end of the program consisted of an infinite loop ('goto \$'), to leave the LED turned on.

Here are the key parts of the assembler code from [mid-range assembler lesson 1](#):

```

                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4 Mhz int clock
__CONFIG      _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

;***** RESET VECTOR *****
RESET        CODE        0x0000        ; processor reset vector
                ; calibrate internal RC oscillator
                call        0x03FF        ; retrieve factory calibration value
                banksel    OSCCAL        ; (stored at 0x3FF as a retlw k)
                movwf      OSCCAL        ; then update OSCCAL

;***** MAIN PROGRAM *****

;***** Initialisation
start
                ; configure port
                movlw      ~(1<<GP1)    ; configure GP1 (only) as an output
                banksel    TRISIO
                movwf      TRISIO

;***** Main code
                ; turn on LED
                banksel    GPIO
                bsf        GPIO,GP1     ; set GP1 high

                ; loop forever
                goto       $

```

XC8 implementation

You should start by creating a new XC8 project, as we did in [baseline C lesson 1](#).

When you run the Project Wizard (or “New Project wizard” if you are using MPLAB X), select the PIC12F629 as the device, and XC8 as the compiler (“Microchip XC8 ToolSuite” in MPLAB 8, or “XC8” in MPLAB X).

Create an empty ‘.c’ source file in your project folder, in the same way as in [baseline C lesson 1](#).

Open it in the MPLAB text editor, and you're ready to start coding!

As usual, you should include a comment block at the start of each program or module. Most of the information in the comment block should be much the same, regardless of the programming language used, since it relates to what this application is, who wrote it, dependencies and the assumed environment, such as pin assignments. However, when writing in C, it is a good idea to state which compiler has been used, since, as we have seen for data types, C code for microcontrollers is not necessarily easily portable.

So we might use something like:

```

/*****
*
*   Filename:      MC_L1-Turn_on_LED-HTC.c
*   Date:         8/6/12
*   File Version: 1.2
*
*   Author:       David Meiklejohn
*   Company:      Gooligum Electronics
*
*****
*
*   Architecture: Mid-range PIC
*   Processor:    12F629
*   Compiler:     MPLAB XC8 v1.00 (Free mode)
*
*****
*
*   Files required: none
*
*****
*
*   Description:   Lesson 1, example 1
*
*   Turns on LED. LED remains on until power is removed.
*
*****
*
*   Pin assignments:
*       GP1 = indicator LED
*
*****/

```

Note that, as we did our previous assembler code, the processor architecture and device are specified in the comment block. This is important for the XC8 compiler, as there is no way to specify the device in the code; i.e. there is no equivalent to the MPASM ‘list p=’ or ‘processor’ directives. Instead, the processor is specified in the IDE (MPLAB), or as a command-line option.

Most of the symbols relevant to specific processors are defined in header files. But instead of including a specific file, as we would do in assembler, it is normal to include a single “catch-all” file: “xc.h” (or “htc.h”). This file identifies the processor being used, and then calls other header files as appropriate. So our next line, which should be at the start of every XC8 program, is:

```
#include <xc.h>
```

The processor can be configured with a series of “*configuration pragmas*”, such as:

```
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
#pragma config MCLRE = ON, CP = OFF, CPD = OFF, BOREN = OFF, WDTE = OFF
#pragma config PWRTE = OFF, FOSC = INTRCIO
```

Or, you can use the ‘__CONFIG’ macro, in a very similar way to the __CONFIG directive in MPASM:

```
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
        PWRTE_OFF & FOSC_INTRCIO);
```

The symbols are the same in both, but note that the pragma uses '=' (with optional spaces) between each setting, such as 'MCLR', and its value, such as 'ON', while the macro uses '_' (with no spaces)⁴. To see which symbols to use for a given PIC, you need to consult the "pic_chipinfo.html" file, in the "docs" directory within the compiler install directory.

As with most C compilers, the entry point for "user" code is a function called 'main()'.

So an XC8 program will look like:

```
void main()
{
    ;    // user code goes here
}
```

Declaring main() as void isn't strictly necessary, since any value returned by main() is only relevant when the program is being run by an operating system which can act on that return value, but of course there is no operating system here. Similarly it would be more "correct" to declare main() as taking no parameters (i.e. main(void)), since there is no operating system to pass any parameters to the program. How you declare main() is really a question of personal style.

At the start of our assembler programs, we've always loaded the OSCCAL register with the factory calibration value (although it is only necessary when using the internal RC oscillator). There is no need to do so when using XC8; the default start-up code, which runs before main(), loads OSCCAL for us.

XC8 makes the PIC's special function registers, such as TRISIO, available as variables.

To load the TRISIO register with 111101b, it is simply a matter of:

```
TRISIO = 0b111101;    // configure GP1 (only) as an output
```

Alternatively this could be expressed as:

```
TRISIO = ~(1<<1);    // configure GP1 (only) as an output
```

Individual bits, such as GP1, can be accessed through bit-fields defined in the header files.

For example, the "pic12f629.h" file header file defines a union called GPIObits, containing a structure with bit-field members GP0, GP1, etc.

So, to set GP1 to '1', we can write:

```
GPIObits.GP1 = 1;    // set GP1 high
```

[Baseline assembler lesson 2](#) explained that setting or clearing a single pin in this way is a "read-modify-write" ("rmw") operation, which may lead to problems.

To avoid any potential for such rmw problems, we can load the value 000010b into GPIO (setting bit 1, and clearing all the other bits), with:

```
GPIO = 0b000010;    // set GP1 high
```

That's not likely to be necessary in this simple example, where only one pin on the GPIO port is being used as an output, and where it is not being changed rapidly – so for simplicity we'll go ahead and set the pin individually, with "GPIObits.GP1 = 1" in this example. Nevertheless it's best to be aware of the potential for rmw issues, and later examples will illustrate ways to avoid it.

⁴ Although the '__CONFIG' macro is now (as of XC8 v1.10) considered to be a "legacy" feature, it is still supported and we will continue to use it in these tutorials (the examples were originally written for HI-TECH C).

Finally, we need to loop forever. There are a number of C constructs that could be used for this, but one that's as good as any is:

```
for (;;)
{
    // loop forever
    ;
}
```

Complete program

Here is the complete code to turn on an LED on GP1:

```

/*****
 *
 * Description: Lesson 1, example 1
 *
 * Turns on LED. LED remains on until power is removed.
 *
 *****/
 *
 * Pin assignments:
 * GP1 = indicator LED
 *
 *****/
#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    TRISIO = ~(1<<1); // configure GP1 (only) as an output
    GPIObits.GP1 = 1; // set GP1 high

    //*** Main loop
    for (;;)
    {
        // loop forever
        ;
    }
}

```

Building the project

Whether you use MPLAB 8 or MPLAB X, the process of compiling and linking your code (making or building your project) is essentially the same as for an assembler project.

To compile the source code in MPLAB 8, select “Project → Build”, press F10, or click on the “Build” toolbar button. This compiles all the source files which have changed, and links the resulting object files and any library functions, creating an output ‘.hex’ file, which can then be programmed into the PIC as normal

(see [baseline assembler lesson 1](#)). The other Project menu item or toolbar button, “Rebuild”, is equivalent to the MPASM “Build All”, recompiling all your source files, regardless of whether they have changed.

Building an XC8 project in MPLAB X is exactly the same as for a MPASM assembler project: click on the “Build” or “Clean and Build” toolbar button, or select the equivalent items in the “Run” menu, to compile and link your code. When it builds without errors and you are ready to program your code into your PIC, select the “Run → Run Main Project” menu item, click on the “Make and Program Device” toolbar button, or simply press F6.

Example 2: Flashing an LED (20% duty cycle)

In [mid-range assembler lesson 1](#), we used the same circuit as above, but made the LED flash by toggling the GP1 output. The delay was created by an in-line busy-wait loop. [Mid-range assembler lesson 2](#) showed how to move the delay loop into a subroutine, and to generalise it, so that the delay is passed as a parameter to the routine, in W. This was demonstrated by a program which flashed the LED at 1 Hz, with a duty cycle of 20%, by turning it on for 200 ms and then off for 800 ms, before repeating.

Here is the main loop from the assembler code from that lesson:

```
main_loop
    ; turn on LED
    banksel GPIO
    movlw    1<<GP1          ; set GP1
    movwf   GPIO
    ; delay 0.2 s
    movlw   .20              ; delay 20 x 10 ms = 200 ms
    call    delay10
    ; turn off LED
    clrf    GPIO             ; (clearing GPIO clears GP1)
    ; delay 0.8 s
    movlw   .80              ; delay 80 x 10ms = 800ms
    call    delay10

    ; repeat forever
    goto   main_loop
```

XC8 implementation

We’ve seen how to turn on the LED on GP1, with:

```
GPIObits.GP1 = 1;          // set GP1
```

or

```
GPIO = 0b000010;         // set GP1 (bit 1 of GPIO)
```

And of course, to turn the LED off, it is simply:

```
GPIObits.GP1 = 0;        // clear GP1
```

or

```
GPIO = 0;                // (clearing GPIO clears GP1)
```

These statements can easily be placed within an endless loop, to repeatedly turn the LED on and off. All we need to add is a delay.

XC8 provides a built-in function, ‘_delay(n)’, which creates a delay ‘n’ instruction clock cycles long. The maximum possible delay depends on which PIC you are using, but it is a little over 50,000,000 cycles. With a 4 MHz processor clock, corresponding to a 1 MHz instruction clock, that’s a maximum delay of a little over 50 seconds.

The compiler also provides two macros: ‘`__delay_us()`’ and ‘`__delay_ms()`’, which use the ‘`_delay(n)`’ function create delays specified in μs and ms respectively. To do so, they reference the symbol “`_XTAL_FREQ`”, which you must define as the processor oscillator frequency, in Hertz.

Since our PIC12F629 is running at 4 MHz, we have:

```
#define _XTAL_FREQ 4000000 // oscillator frequency for _delay()
```

Then, to generate a 200 ms delay, we can write:

```
    __delay_ms(200); // stay on for 200 ms
```

Complete program

Putting these delay macros into the main loop, we have:

```

/*****
 * Description: Lesson 1, example 2
 *
 * Flashes an LED at approx 1 Hz, with 20% duty cycle
 * LED continues to flash until power is removed.
 *
 *****/
 * Pin assignments:
 * GP1 = flashing LED
 *
 *****/
#include <xc.h>

#define _XTAL_FREQ 4000000 // oscillator frequency for _delay()

/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    TRISIO = 0b111101; // configure GP1 (only) as an output

    /*** Main loop
    for (;;)
    {
        GPIO = 0b000010; // turn on LED on GP1 (bit 1)
        __delay_ms(200); // stay on for 200 ms
        GPIO = 0; // turn off LED (clearing GPIO clears GP1)
        __delay_ms(800); // stay off for 800 ms
    } // repeat forever
}

```

Example 3: Flashing an LED (50% duty cycle)

The LED flashing example in [mid-range assembler lesson 1](#) used an XOR operation to flip the GP1 bit every 500 ms, creating a 1 Hz flash with a 50% duty cycle.

The read-modify-write problem revisited

As discussed in that lesson, any operation which reads an output (or part-output) port, modifies the value read, and then writes it back, can lead to unexpected results. This is because, when a port is read, it is the value at the pins that is read, not necessarily the value that was written to the output latches. And that's a problem if, for example, you have written a '1' to an output pin, which, because it is being externally loaded (or, more usually, it hasn't finished going high yet, because of a capacitive load on the pin), it reads back as a '0'. When the operation completes, that output bit would be written back as a '0', and the output pin sent low instead of high – not what it is supposed to be.

This can happen with any instruction which reads the current value of a register when updating it. That includes logic operations such as XOR, but also arithmetic operations (add, subtract), rotate instructions, and increment and decrement operations. And crucially, it also includes the bit set and clear instructions.

You may think that the instruction `'bsf GPIO, 1'` will only affect GP1, but in fact that instruction reads the whole of GPIO, sets bit 1, and then writes the whole of GPIO back again.

Consider the sequence:

```
bsf    GPIO, 1
bsf    GPIO, 2
```

Assuming that GP1 and GP2 are both initially low, the first instruction will attempt to raise the GP1 pin high. However, the first instruction writes to GPIO at the end of the instruction cycle, while the second instruction reads the port pins toward the start of the following instruction cycle. That doesn't leave much time for GP1 to be pulled high, against whatever capacitance is loading the pin. If it hasn't gone high enough by the time the second `'bsf'` instruction reads the pins, it will read as a '0', and it will then be written back as a '0' when the second `'bsf'` writes to GPIO. The potential result is that, instead of both GP1 and GP2 being set high, as you would expect, it is possible that only GP2 will be set high, while the GP1 pin remains low, and the GP1 bit holds a '0'.

This problem is sometimes avoided by placing `'nop'` instructions between successive read-modify-write operations, but as we've seen in the assembler lessons, a more robust solution is to use a shadow register.

So why revisit this topic, in a lesson on C programming?

When you use a statement like `'GPIObits.GP1 = 1'` in XC8, the compiler translates those statements into corresponding bit set or clear instructions, which may lead to read-modify-write problems.

Note: Any C statements which directly modify individual port bits may be subject to read-modify-write considerations.

There was no problem with using these types of statements in the examples above, where only a single pin is being used and there are lengthy delays between changes.

But you should be aware that a sequence such as:

```
GPIObits.GP1 = 1;
GPIObits.GP2 = 1;
```

may in fact result in GP1 being cleared and only GP2 being set high.

To avoid such problems, shadow variables can be used in C programs, in much the same way that they are used in assembly language.

Here is the main code from the program presented in [mid-range assembler lesson 2](#):

```

; configure port
movlw  ~(1<<GP1)      ; configure GP1 (only) as an output
banksel TRISIO
movwf  TRISIO

        clrfsf  sGPIO      ; start with shadow GPIO zeroed

;***** Main loop
main_loop
; toggle LED
movf   sGPIO,w         ; get shadow copy of GPIO
xorlw  1<<GP1          ; toggle bit corresponding to GP1
movwf  sGPIO           ; in shadow register
banksel GPIO           ; and write to GPIO
movwf  GPIO

; delay 500 ms -> 1 Hz flashing at 50% duty cycle
movlw  .50
pagesel delay10        ; delay 50 x 10 ms = 500 ms
call   delay10

; repeat forever
pagesel main_loop
goto  main_loop

```

XC8 implementation

To toggle GP1, you could use the statement:

```
GPIObits.GP1 = ~GPIObits.GP1;
```

or:

```
GPIObits.GP1 = !GPIObits.GP1;
```

This statement is also supported:

```
GPIObits.GP1 = GPIObits.GP1 ? 0 : 1;
```

It works because single-bit bit-fields, such as GP1, hold either a '0' or '1', representing 'false' or 'true' respectively, and so can be used directly in a conditional expression like this.

However, since these statements modify individual bits in GPIO, to avoid potential read-modify-write issues we'll instead use a shadow variable, which can be declared and initialised with:

```
uint8_t    sGPIO = 0;           // shadow copy of GPIO
```

This makes it clear that the variable is an unsigned, eight-bit integer. We could have declared this as an 'unsigned char', or simply 'char' (because 'char' is unsigned by default), but you can make your code clearer and more portable by using the C99 standard integer types defined in the "stdint.h" header file.

To define these standard integer types, add this line toward the start of your program:

```
#include <stdint.h>
```

The variable declaration could be placed within the `main()` function, which is what you should do for any variable that is only accessed within `main()`. However, a variable such as a shadow register may need to be accessed by other functions. For example, it's quite common to place all of your initialisation code into a function called `init()`, which might initialise the shadow register variables as well as the ports, and your `main()` code may also need to access them. It is often best to define such variables as global (or "external") variables toward the start of your code, before any functions, so that they can be accessed throughout your program.

But remember that, to make your code more maintainable and to minimise data memory use, you should declare any variable which is only used by one function, as a local variable within that function.

We'll see examples of that later, but in this example we'll define `sGPIO` as a global variable.

Flipping the shadow copy of **GP1** and updating **GPIO**, can then be done by:

```
sGPIO ^= 1<<1;           // flip shadow bit corresponding to GP1
GPIO = sGPIO;           // write to GPIO
```

Complete program

Here is how the XC8 code to flash a LED on **GP1**, with a 50% duty cycle, fits together:

```

/*****
 *
 * Description: Lesson 1, example 3
 *
 * Flashes a LED at approx 1 Hz.
 * LED continues to flash until power is removed.
 *
 *****/
 *
 * Pin assignments:
 * GP1 = flashing LED
 *
 *****/

#include <xc.h>
#include <stdint.h>

#define _XTAL_FREQ 4000000 // oscillator frequency for _delay()

//***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

//***** GLOBAL VARIABLES *****/
uint8_t sGPIO = 0; // shadow copy of GPIO

//***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    TRISIO = ~(1<<1); // configure GP1 (only) as an output

```

```

//*** Main loop
for (;;)
{
    // toggle LED on GP1
    sGPIO ^= 1<<1;           // flip shadow bit corresponding to GP1
    GPIO = sGPIO;           // write to GPIO

    // delay 500 ms
    __delay_ms(500);

} // repeat forever
}

```

Comparisons

Although this is a very small, simple application, it is instructive to compare the source code size (lines of code⁵) and resource utilisation (program and data memory usage) for this C version with the assembler version of this example from [mid-range assembler lesson 2](#).

Source code length is a rough indication of how difficult or time-consuming a program is to write. We expect that C code is easier and quicker to write than assembly language, but that a C compiler (especially one with optimisation disabled, as XC8 is, when running in “Free mode”) will produce code that is bigger or uses memory less efficiently than hand-crafted assembly. But is this true?

It’s also interesting to see whether the delay function provided by the C compiler generates accurately-timed delays, and how its accuracy compares with our assembler version.

MPLAB correctly reports the memory usage for assembler and XC8 projects, and the MPLAB simulator⁶ can be used to accurately measure the time between LED flashes – ideally it would be exactly 1.000000 seconds, and the difference from that gives us the overall timing error.

Here is the resource usage and accuracy summary for the “Flash an LED at 50% duty cycle” programs. The resource usage and accuracy of the baseline (12F509) versions of this example, from [baseline assembler lesson 3](#) and [baseline C lesson 2](#), is also given for comparison:

Flash_LED-50p

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)		Delay accuracy (timing error)	
	12F629	12F509	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	27	28	29	34	4	4	0.15%	0.15%
XC8 (Free mode)	11	11	35	36	4	4	0.0024%	0.0024%

The assembler version called the delay routine as an external module, so it’s quite comparable with the C programs which make use of built-in delay functions. Nevertheless, the assembly language source code is around three times as long as the C version! This illustrates how much more compact C code can be.

As for C being less efficient – the XC8 version is only a little larger than the assembler version, despite having most compiler optimisations disabled in “Free mode”. This is largely because the built-in delay code (which, as we can see is highly accurate!) is optimised, but it does show that C is not necessarily inherently inefficient.

⁵ ignoring whitespace, comments, and “unnecessary” lines such as the redefinition of pin names in the CCS C examples

⁶ a topic for a future tutorial?

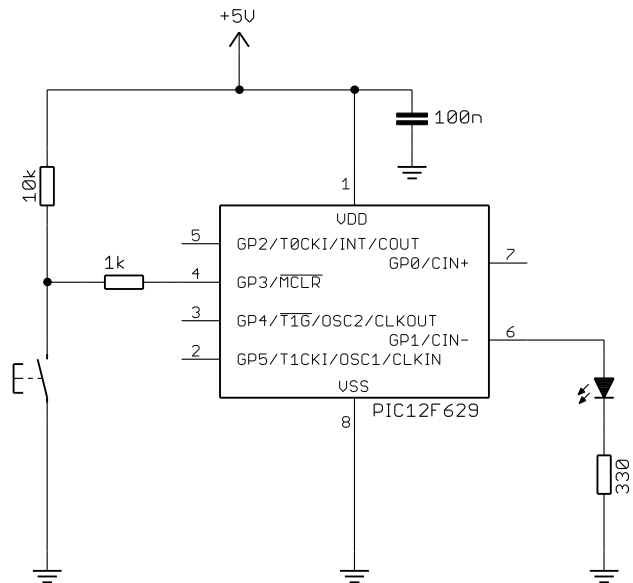
Example 4: Reading Digital Inputs

[Mid-range assembler lesson 3](#) introduced digital inputs, using a pushbutton switch in the simple circuit shown on the right.

It's the same circuit as in the earlier examples (you can leave your board configured the same way as before), but now we'll use the pushbutton to drive a digital input (GP3), instead of as a reset switch.

The 10 kΩ resistor normally holds the GP3 input high, until the pushbutton is pressed, pulling the input low.

Note: if you are using a PICkit 2 programmer, you must enable '3-State on "Release from Reset"', as described in [mid-range assembler lesson 3](#), to allow the pushbutton to pull GP3 low when pressed.



As an initial example, the pushbutton input was copied to the LED output, so that the LED was on, whenever the pushbutton is pressed.

In pseudo-code, the operation is:

```
do forever
  if button down
    turn on LED
  else
    turn off LED
end
```

The assembler code we used to implement this, using a shadow register, was:

```
; configure port
movlw  ~(1<<GP1)      ; configure GP1 (only) as an output
banksel TRISIO       ; (GP3 is an input)
movwf  TRISIO

;***** Main loop
main_loop
  ; turn on LED only if button pressed
  clrf  sGPIO         ; assume button up -> LED off
  banksel GPIO
  btfss GPIO,GP3     ; if button pressed (GP3 low)
  bsf   sGPIO,GP1    ; turn on LED

  movf  sGPIO,w      ; copy shadow to GPIO
  movwf GPIO

  goto  main_loop    ; repeat forever
```

XC8 implementation

To copy a value from one bit to another, e.g. GP3 to GP1, using XC8, can be done as simply as:

```
GPIObits.GP1 = GPIObits.GP3;           // copy GP3 to GP1
```

But that won't do quite what we want; given that GP3 goes low when the button is pressed, simply copying GP3 to GP1 would lead to the LED being on when the button is up, and on when it is pressed – the opposite of the required behaviour.

We can address that by inverting the logic:

```
GPIObits.GP1 = !GPIObits.GP3;         // copy !GP3 to GP1
```

or

```
GPIObits.GP1 = GPIObits.GP3 ? 0 : 1;   // copy !GP3 to GP1
```

This works well in practice, but to allow a valid comparison with the assembly source above, which uses a shadow register, we should not use statements which modify individual bits in GPIO. Instead we should write an entire byte to GPIO at once.

For example, we could write:

```
if (GPIObits.GP3 == 0) // if button pressed
    GPIO = 0b000010;   // turn on LED
else
    GPIO = 0;          // else turn off LED
```

However, this can be written much more concisely using C's conditional expression:

```
GPIO = GPIObits.GP3 ? 0 : 0b000010; // if GP3 high, clear GP1, else set GP1
```

It may seem a little obscure, but this is exactly the type of situation the conditional expression is intended for.

Complete program

Here is the complete XC8 code to turn on a LED when a pushbutton is pressed:

```

/*****
 *
 * Description: Lesson 1, example 4
 *
 * Demonstrates reading a switch
 *
 * Turns on LED when pushbutton is pressed
 *
 *****/
 *
 * Pin assignments:
 * GP1 = indicator LED
 * GP3 = pushbutton switch (active low)
 *
 *****/
#include <xc.h>

/***** CONFIGURATION *****/
// int reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
        PWRTE_OFF & FOSC_INTRCIO);

```

```

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    TRISIO = ~(1<<1);           // configure GP1 (only) as an output

    /*** Main loop
    for (;;)
    {
        // turn on LED only if button pressed
        GPIO = GPIObits.GP3 ? 0 : 0b000010;    // if GP3 high, clear GP1,
                                                // else set GP1
    }
}

```

Note that the processor configuration has been changed to disable the external $\overline{\text{MCLR}}$ reset, so that GP3 is available as an input.

Comparisons

Here is the resource usage summary for the “Turn on LED when pushbutton pressed” programs:

PB_LED

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	21	18	16	13	1	1
XC8 (Free mode)	6	6	29	29	2	2

At only six lines, the C source code is amazingly succinct – thanks mainly to the use of C’s conditional expression (?:).

Example 5: Switch Debouncing

[Mid-range assembler lesson 3](#) included a discussion of the switch contact bounce problem, and various hardware and software approaches to addressing it.

The problem was illustrated by an example application, using the circuit from example 4 (above), where the LED is toggled each time the pushbutton is pressed. If the switch is not debounced, the LED toggles on every contact bounce, making it difficult to control.

The most sophisticated software debounce method presented in that lesson was a counting algorithm, where the switch is read (*sampled*) periodically (e.g. every 1 ms) and is only considered to have definitely changed state if it has been in the new state for some number of successive samples (e.g. 10), by which time it is considered to have settled.

The algorithm was expressed in pseudo-code as:

```
count = 0
while count < max_samples
  delay sample_time
  if input = required_state
    count = count + 1
  else
    count = 0
end
```

It was implemented in assembler as follows:

```

; wait for button press, debounce by counting:
db_dn  movlw  .13          ; max count = 10ms/768us = 13
        movwf  db_cnt
        clrf   dcl
dn_dly  incfsz  dcl,f      ; delay 256x3 = 768 us.
        goto  dn_dly
        btfsc  GPIO,GP3   ; if button up (GP3 high),
        goto  db_dn       ; restart count
        decfsz db_cnt,f    ; else repeat until max count reached
        goto  dn_dly
```

This code waits for the button to be pressed (GP3 being pulled low), by sampling GP3 every 768 μ s and waiting until it has been low for 13 times in succession – approximately 10 ms in total.

XC8 implementation

To implement the counting debounce algorithm (above) using XC8, the pseudo-code can be translated almost directly into C:

```

db_cnt = 0;
while (db_cnt < 10)
{
    __delay_ms(1);
    if (GPIObits.GP3 == 0)
        db_cnt++;
    else
        db_cnt = 0;
}
```

where the debounce counter variable has been declared as:

```
uint8_t    db_cnt;          // debounce counter
```

Note that, because this variable is only used locally (other functions would never need to access it), it should be declared within `main()`.

Whether you modify this code to make it shorter is largely a question of personal style. Compressed C code, using a lot of “clever tricks” can be difficult to follow.

But note that the `while` loop above is equivalent to the following `for` loop:

```

for (db_cnt = 0; db_cnt < 10;)
{
    __delay_ms(1);
    if (GPIObits.GP3 == 0)
        db_cnt++;
    else
        db_cnt = 0;
}
```

That suggests restructuring the code into a traditional `for` loop, as follows:

```
for (db_cnt = 0; db_cnt <= 10; db_cnt++)
{
    __delay_ms(1);
    if (GPIObits.GP3 == 1)
        db_cnt = 0;
}
```

In this case, the debounce counter is incremented every time around the loop, regardless of whether it has been reset to zero within the loop body. For that reason, the end of loop test has to be changed from '`<`' to '`<=`', so that the number of iterations remains the same.

Alternatively, the loop could be written as:

```
for (db_cnt = 0; db_cnt < 10;)
{
    __delay_ms(1);
    db_cnt = (GPIObits.GP3 == 0) ? db_cnt+1 : 0;
}
```

However the previous version seems easier to understand.

Complete program

Here is the complete XC8 code to toggle an LED when a pushbutton is pressed, including the debounce routines for button-up and button-down:

```
/******
 *
 * Description: Lesson 1, example 5
 *
 * Demonstrates use of counting algorithm for debouncing
 *
 * Toggles LED when pushbutton is pressed then released,
 * using a counting algorithm to debounce switch
 *
 ******
 *
 * Pin assignments:
 * GP1 = indicator LED
 * GP3 = pushbutton switch (active low)
 *
 ******/

#include <xc.h>
#include <stdint.h>

#define _XTAL_FREQ 4000000 // oscillator frequency for _delay()

/***** CONFIGURATION *****/
// int reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
        PWRTE_OFF & FOSC_INTRCIO);

/***** GLOBAL VARIABLES *****/
uint8_t sGPIO; // shadow copy of GPIO
```

```

/***** MAIN PROGRAM *****/
void main()
{
    uint8_t    db_cnt;           // debounce counter

    /*** Initialisation

    // configure port
    GPIO = 0;                   // start with LED off
    sGPIO = 0;                  // update shadow
    TRISIO = ~(1<<1);          // configure GP1 (only) as an output

    /*** Main loop
    for (;;)
    {
        // wait for button press, debounce by counting:
        for (db_cnt = 0; db_cnt <= 10; db_cnt++)
        {
            __delay_ms(1);      // sample every 1 ms
            if (GPIObits.GP3 == 1) // if button up (GP3 high)
                db_cnt = 0;     // restart count
        }                       // until button down for 10 successive reads

        // toggle LED on GP1
        sGPIO ^= 0b000010;     // toggle shadow GP1
        GPIO = sGPIO;          // write to GPIO

        // wait for button release, debounce by counting:
        for (db_cnt = 0; db_cnt <= 10; db_cnt++)
        {
            __delay_ms(1);      // sample every 1 ms
            if (GPIObits.GP3 == 0) // if button down (GP3 low)
                db_cnt = 0;     // restart count
        }                       // until button up for 10 successive reads

    } // repeat forever
}

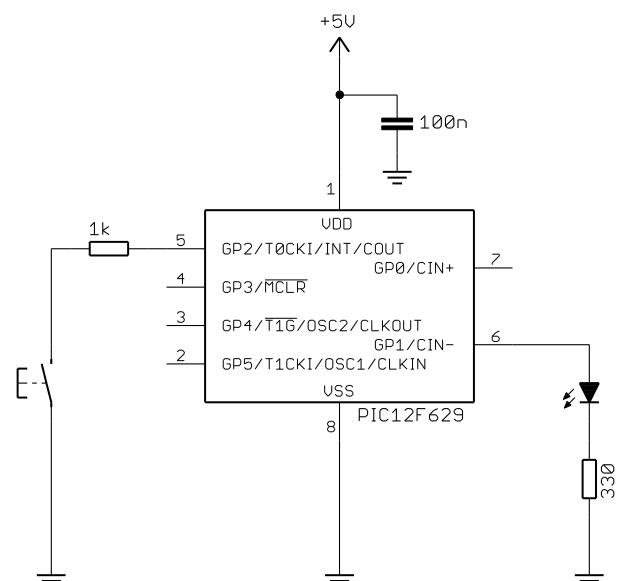
```

Example 6: Internal (Weak) Pull-ups

As discussed in [mid-range assembler lesson 3](#), many PICs include internal “weak pull-ups”, which can be used to pull floating inputs (such as an open switch) high. They perform the same function as external pull-up resistors, pulling an input high when a connected switch is open, but only supplying a small current; not enough to present a problem when a closed switch grounds the input.

This means that, on pins with a weak pull-up, it is possible to directly connect switches between an input pin and ground, as shown on the right.

Unfortunately, there is no internal pull-up on the 12F629’s GP3 pin, so to demonstrate their use we need to use a different input pin, which is why the switch is connected to GP2 here.



The [Gooligum training board](#) already has a pushbutton switch connected to GP2 as shown, but you should ensure that jumper JP7 is not closed, so that there is no external pull-up in place.

If you are using the Microchip demo board, you will need to supply your own pushbutton and connect it between GP2 (pin 9 of the 14-pin header) and ground (pin 14 on the header).

To enable the weak pull-ups, clear the $\overline{\text{GPPU}}$ bit in the OPTION register.

In the example assembler program from [mid-range lesson 3](#), this was done by:

```
bcf    OPTION_REG,NOT_GPPU    ; enable weak pull-ups (global)
```

Unlike the baseline PICs, the weak pull-ups on mid-range devices individually selectable; to enable a pull-up, the corresponding bit in the WPU register must be set.

By default (after a power-on reset) every bit in WPU is set, so to enable a pull-up on only a single pin, the remaining bits in WPU must be cleared.

This was done, in the assembler example in [mid-range lesson 3](#), by:

```
movlw  1<<GP2                ; select pull-up on GP2 only
movwf  WPU
```

XC8 implementation

The XC8 compiler makes the individual bits in the OPTION register available as bit-fields, so to clear $\overline{\text{GPPU}}$, without affecting any of the other OPTION bits, we can simply write:

```
OPTION_REGbits.nGPPU = 0;    // enable weak pull-ups (global)
```

Note again you should look at the header file for your PIC (“pic12f629.h” in this case) to check the name of the bit-field associated with the register bit you wish to access. It is not necessarily the same as the symbol defined in the assembler include file – for example, the XC8 include file defines the bit-field ‘nGPPU’, while the MPASM include file defines the symbol ‘NOT_GPPU’, both referring to the $\overline{\text{GPPU}}$ bit.

Similarly, the WPU register is accessible through a variable named ‘WPU’, so to enable the pull-up on GP2 (and disable all the other pull-ups) can write:

```
WPU = 1<<2;                  // select pull-up on GP2 only
```

With these additions, the initialisation code then becomes:

```
// configure port
OPTION_REGbits.nGPPU = 0;    // enable weak pull-ups (global)
WPU = 1<<2;                  // enable pull-up on GP2 only
GPIO = 0;                    // start with LED off
sGPIO = 0;                   // update shadow
TRISIO = ~(1<<1);           // configure GP1 (only) as an output
```

The rest of the program is then the same as before (example 5, above), except for testing GP2 instead of GP3.

Comparisons

Here is the resource usage summary for the “toggle a LED using weak pull-ups” programs:

Toggle_LED+WPU

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	47	43	40	36	3	3
XC8 (Free mode)	23	22	97	94	3	3

Although the difference is less pronounced than in the simpler, earlier examples, the C source code continues to be less than half the length of the assembly version, while the (unoptimised) code generated by the XC8 compiler continues to be more than twice the size of the hand-written assembly language version.

Summary

Overall, we have seen that basic digital I/O operations can be expressed succinctly using C, leading to significantly shorter source code than assembly language, as illustrated by the comparisons we have done in this lesson: the C code is typically only half, or less, as long as the corresponding assembler source code.

It could be argued that, because the C code is significantly shorter than corresponding assembler code, with the program structure more readily apparent, C programs are more easily understood, faster to write, and simpler to debug, than assembler.

So why use assembler? One argument is that, because assembler is closer to the hardware, the developer benefits from having a greater understanding of exactly what the hardware is doing; there are no unexpected or undocumented side effects, no opportunities to be bitten by bugs in built-in or library functions. But that argument applies more to other compilers than it does to XC8, which exposes all the PIC’s registers as variables, and the programmer has to directly modify the register contents in much the same way as would be done in assembler.

However, C compilers usually generate code which occupies more program memory and uses more data memory than for corresponding hand-written assembler programs⁷.

Since the C compilers consistently use more resources than assembler (for equivalent programs), there comes a point, as programs grow, that a C program will not fit into a particular PIC, while the same program would have fit if it had been written in assembler. In that case, the choice is to write in assembler, or use a more expensive PIC. For a one-off project, a more expensive chip probably makes sense, whereas for volume production, using resources efficiently by writing in assembler may be the right choice.

In the [next lesson](#) we’ll see how to use XC8 to configure and access Timer0.

⁷ It’s a little unfair to draw this conclusion, based on a compiler (XC8) running with optimisation disabled (in “Free mode”). However, this statement remains true when the PICC-Lite compiler (sadly, no longer available), with full optimisation enabled, is used to implement these examples.

Introduction to PIC Programming

Programming Mid-Range PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 2: Using Timer 0

As we saw in the [previous lesson](#), C can be a viable choice for programming digital I/O operations on mid-range (14-bit) PICs, although we also saw that programs written in C can consume significantly more memory (a limited resource on these tiny MCUs) than equivalent programs written in assembler.

This lesson revisits the material from [mid-range assembler lesson 4](#) on the Timer0 module: using it to time events, to maintain the timing of a background task, for switch debouncing, and as a counter.

Selected examples are re-implemented using Microchip's XC8 compiler¹ (running in "Free mode"), introduced in [lesson 1](#). We'll also see the C equivalents of some of the features covered in [mid-range assembler lesson 5](#), including macros.

In summary, this lesson covers:

- Configuring Timer0 as a timer or counter
- Accessing Timer0
- Using Timer0 for switch debouncing
- Using C macros

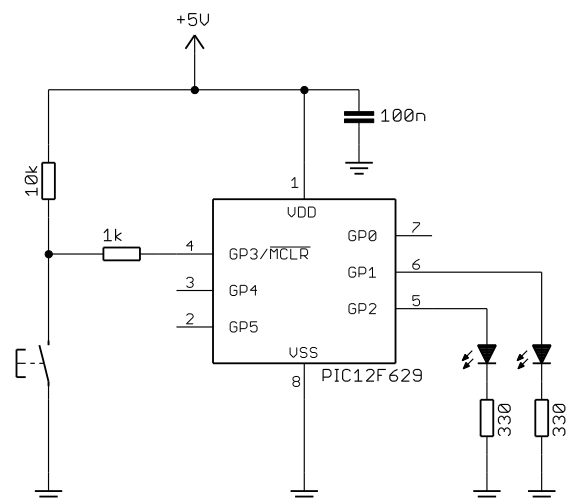
Note that this tutorial series assumes a working knowledge of the C language; it does **not** attempt to teach C.

Example 1: Using Timer0 as an Event Timer

To demonstrate how Timer0 can be used to measure elapsed time, [mid-range assembler lesson 4](#) included a "reaction timer" game, using the circuit on the right.

To implement this circuit using the [Gooligum training board](#), connect jumpers JP3, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP1 and GP2.

If you are using Microchip's Low Pin Count Demo Board, you will need to connect LEDs to GP1 and GP2, as described in [baseline assembler lesson 1](#).



¹ Available as a free download from www.microchip.com.

The pushbutton has to be pressed as quickly as possible after the LED on GP2, indicating ‘start’ is lit. If the button is pressed within a predefined reaction time, the LED on GP1 is lit, to indicate ‘success’. Thus, we need to measure the elapsed time between indicating ‘start’ and detecting a pushbutton press.

An ideal way to do that is to use Timer0, in its timer mode (clocked by the PIC’s instruction clock, which in this example is 1 MHz).

Ideally, to make a better “game”, the delay before the ‘start’ LED is lit would be random, but in this simple example, a fixed delay is used.

The program flow can be illustrated in pseudo-code as:

```
do forever
    clear both LEDs
    delay 2 sec
    indicate start
    clear timer
    wait up to 1 sec for button press
    if button pressed and elapsed time < 200ms
        indicate success
    delay 1 sec
end
```

To use Timer0 to measure the elapsed time, we need to extend its range (normally limited to 65 ms) by adding a counter variable, which is incremented each time the timer overflows (or reaches a certain value). In the example in [mid-range lesson 4](#), Timer0 is configured so that it is clocked every 32 μ s, by using the 1 MHz instruction clock with a 1:32 prescaler. After 250 counts, 8 ms ($250 \times 32 \mu$ s) will have elapsed; this is used to increment a counter, which effectively measures time in 8 ms intervals. When the button is pressed, this “8 ms counter” can then be checked, to see whether the maximum reaction time has been exceeded.

As explained in [mid-range lesson 4](#), to select timer mode, with a 1:32 prescaler, we must clear the TOCS and PSA bits, in the OPTION register, and set the PS<2:0> bits to 100. This was done by:

```
    movlw    b'11000100'    ; configure Timer0:
    ; --0-----            timer mode (TOCS = 0)
    ; ----0---             prescaler assigned to Timer0 (PSA = 0)
    ; -----100           prescale = 32 (PS = 100)
    banksel OPTION_REG    ; -> increment TMR0 every 32us
    movwf   OPTION_REG
```

Here is the main assembler code we had used to implement the button press / timing test routine:

```
    ; wait up to 1 sec for button press
    clrf    cnt_8ms        ; clear timer (8 ms counter)
wait1s    ; repeat for 1 sec:
    banksel TMR0
    clrf    TMR0          ; clear Timer0
w_tmr0    ; repeat for 8 ms:
    banksel GPIO
    btfss  BUTTON        ; if button pressed (low)
    goto   wait1s_end    ; finish delay loop immediately
    banksel TMR0
    movf   TMR0,w        ;
    xorlw  8000/32       ; (8 ms at 32 us/tick)
    btfss  STATUS,Z      ;
    goto   w_tmr0
    incf   cnt_8ms,f     ; increment 8 ms counter
    movlw  1000/8        ; (1 sec at 8 ms/count)
    xorwf  cnt_8ms,w
    btfss  STATUS,Z      ;
    goto   wait1s
wait1s_end
```

```

; indicate success if elapsed time < 200 ms
movlw   MAXRT/8           ; if time < max reaction time (8 ms/count)
subwf   cnt_8ms,w
banksel GPIO
btfss   STATUS,C
bsf     SUCCESS           ; turn on success LED

```

(This code is actually taken from [mid-range assembler lesson 5](#))

XC8 implementation

Loading the OPTION register in XC8 can be done by assigning a value to the variable OPTION_REG:

```

OPTION_REG = 0b11000100;           // configure Timer0:
//--0-----           timer mode (T0CS = 0)
//----0---           prescaler assigned to Timer0 (PSA = 0)
//-----100         prescale = 32 (PS = 100)
//                               -> increment every 32 us

```

Note that this has been commented in a way which documents which bits affect each setting, with ‘-’s indicating “don’t care”. For example, we could have instead used ‘OPTION_REG = 0b11010100’, since the value of bit 4, or TOSE, is irrelevant in timer mode.

However, as we’ve seen, XC8 also makes the individual OPTION register bits available as bit-fields defined in the processor header files (such as “pic12f629.h”), so we can instead write this as:

```

// configure Timer0
OPTION_REGbits.T0CS = 0;           // select timer mode
OPTION_REGbits.PSA = 0;           // assign prescaler to Timer0
OPTION_REGbits.PS = 0b100;        // prescale = 32
//                               // -> increment every 32 us

```

Note that, although the PS bits can be accessed as the single-bit fields PS0 to PS2, they are also (much more conveniently) made available through a 3-bit field named PS, as used here.

This is clear and easy to maintain, and less error-prone, because it is easy to mistype a numeric constant, and the compiler cannot warn you if that happens.

On the other hand, a series of single-bit assignments like this requires more program memory than a whole-register assignment. It is also no longer an *atomic* operation, where all the bits are updated at once. This can be an important consideration in some instances², but it is not relevant here. Note also that the remaining bits in the OPTION register are not being explicitly set or cleared; that is ok because in this example we don’t care what values they have.

Which method you use is largely a question of personal style – and you can adapt your style as appropriate. It is often preferable to use symbolic bit names to specify just one or two register bits, but using binary constants if several bits need to be specified at once, especially where some bits need to be set and others cleared (as is the case here), is quite acceptable – assuming that it is clearly commented, as above.

² for example, an interrupt service routine (see [lesson 3](#)) may rely on a peripheral, such as Timer0 here, being fully configured. It would be a problem if the interrupt occurred in the middle of configuration statements like this, with the peripheral only partially configured. These problems can be addressed by selectively disabling and re-enabling interrupts; we’ll see an example of this in [lesson 8](#).

The TMR0 register is accessed through a variable, TMR0, so to clear it, we can write:

```
TMR0 = 0;                // clear timer0
```

and then to wait until 8 ms has elapsed:

```
while (TMR0 < 8000/32) // wait for 8 ms (32 us/tick)
    ;
```

The “wait for button press or one second” routine can then be implemented as:

```
cnt_8ms = 0;
while (BUTTON == 1 && cnt_8ms < 1000/8)
{
    TMR0 = 0;                // clear timer0
    while (TMR0 < 8000/32) // wait for 8 ms (32 us/tick)
        ;
    ++cnt_8ms;                // increment 8 ms counter
}
```

where ‘BUTTON’ has been defined as a symbol for ‘GPIObits.GP3’.

As discussed in [mid-range assembler lesson 5](#), your code will be easier to understand and maintain if you use symbolic names to refer to pins. If your design changes, you can update the definitions in one place (usually placed at the start of your c, or in a header file). Of course, you may also need to modify your initialisation statements, such as ‘TRISIO =’. This is a good reason to keep all your initialisation code in one easily-found place, such as at the start of the program, or in an “init()” function.

Finally, checking elapsed time is simply:

```
if (cnt_8ms < MAXRT/8) // if time < max reaction time (8 ms/count)
    SUCCESS = 1;       // turn on success LED
```

Complete program

Here is the complete reaction timer program, so that you can see how the various parts fit together:

```

/*****
*
* Description:      Lesson 2, example 1
*                  Reaction Timer game.
*
* User must attempt to press button within defined reaction time
* after "start" LED lights. Success is indicated by "success" LED.
*
* Starts with both LEDs unlit.
* 2 sec delay before lighting "start"
* Waits up to 1 sec for button press
* (only) on button press, lights "success"
* 1 sec delay before repeating from start
*
*****
*
* Pin assignments:
* GP1 = success LED
* GP2 = start LED
* GP3 = pushbutton switch (active low)
*
*****/

```

```

#include <xc.h>
#include <stdint.h>

#define _XTAL_FREQ 4000000 // oscillator frequency for _delay()

/***** CONFIGURATION *****/
// int reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define START GPIObits.GP2 // LEDs
#define SUCCESS GPIObits.GP1

#define BUTTON GPIObits.GP3 // pushbutton

/***** CONSTANTS *****/
#define MAXRT 200 // Maximum reaction time in ms

/***** MAIN PROGRAM *****/
void main()
{
    uint8_t cnt_8ms; // counter: increments every 8 ms

    //*** Initialisation

    // configure port
    TRISIO = 0b111001; // configure GP1 and GP2 as outputs

    // configure Timer0
    OPTION_REGbits.T0CS = 0; // select timer mode
    OPTION_REGbits.PSA = 0; // assign prescaler to Timer0
    OPTION_REGbits.PS = 0b100; // prescale = 32
    // -> increment every 32 us

    //*** Main loop
    for (;;)
    {
        // start with both LEDs off
        GPIO = 0;

        // delay 2 sec
        __delay_ms(2000); // delay 2000 ms

        // indicate start
        START = 1; // turn on start LED

        // wait up to 1 sec for button press
        cnt_8ms = 0;
        while (BUTTON == 1 && cnt_8ms < 1000/8)
        {
            TMR0 = 0; // clear timer0
            while (TMR0 < 8000/32) // wait for 8 ms (32 us/tick)
            {
                ++cnt_8ms; // increment 8 ms counter
            }
        }
    }
}

```

```

    // check elapsed time
    if (cnt_8ms < MAXRT/8)      // if time < max reaction time (8 ms/count)
        SUCCESS = 1;           // turn on success LED

    // delay 1 sec
    __delay_ms(1000);           // delay 1000 ms
} // repeat forever
}

```

Comparisons

As we did in [lesson 1](#), we can compare the length of the source code (ignoring comments and white space) versus program and data memory utilisation for this C version with the assembly version from [mid-range lesson 5](#). The stats for the baseline (PIC12F509) versions of this example, from [baseline C lesson 3](#), are also given for comparison:

Reaction_timer

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	58	53	53	55	4	4
XC8 (Free mode)	26	23	87	83	4	4

As expected, the C source code is less than half as long as the assembler source, but the generated C program code is significantly larger (not surprising, given that XC8 does not perform any optimisation when running in “Free mode”).

Example 2: Background Process Timing

As discussed in [mid-range lesson 4](#), one of the key uses of timers is to provide regular timing for “background” processes, while a “foreground” process responds to user signals. Timers are ideal for this, because they continue to run, at a steady rate, regardless of any processing the PIC is doing. On mid-range PICs this is normally done using timer-driven interrupts, which will be covered in the [next lesson](#). However, the non-interrupt method, described in [baseline C lesson 3](#), can still be used, and is covered here mainly for completeness.

The example in [mid-range lesson 4](#) used the circuit above, flashing the LED on GP2 at a steady 1 Hz, while lighting the LED on GP1 whenever the pushbutton is pressed.

The 500 ms delay needed for the 1 Hz flash was derived from Timer0 as follows:

- Using a 4 MHz processor clock, providing a 1 MHz instruction clock and a 1 μ s instruction cycle
- Assigning a 1:32 prescaler to the instruction clock, incrementing Timer0 every 32 μ s
- Resetting Timer0 to zero, as soon as it reaches 125 (i.e. every $125 \times 32 \mu$ s = 4 ms)
- Repeating 125 times, creating a delay of 125×4 ms = 500 ms.

This was implemented by the following code:

```
main_loop
    ; delay 500 ms
    movlw    .125                ; repeat 125 times (125 x 4 ms = 500 ms)
    movwf    dly_cnt

dly500
    banksel  TMR0                ; clear timer0
    clrf     TMR0
w_tmr0    movf     TMR0,w         ; wait for 4 ms
    xorlw    .125                ; (125 ticks x 32 us/tick = 4 ms)
    btfss   STATUS,Z
    goto     w_tmr0
    decfsz  dly_cnt,f           ; end 500 ms delay loop
    goto     dly500

    ; toggle flashing LED
    movf     sGPIO,w
    xorlw    1<<GP2             ; toggle LED on GP2
    movwf    sGPIO              ; using shadow register
    banksel  GPIO
    movwf    GPIO

    ; repeat forever
    goto     main_loop
```

And then the code which responds to the pushbutton was placed within the timer wait loop:

```
w_tmr0
    banksel  GPIO                ; repeat for 4 ms:
    bcf     sGPIO,GP1           ; check and respond to button press
    btfss   GPIO,GP3           ; assume button up -> indicator LED off
    bsf     sGPIO,GP1           ; if button pressed (GP3 low)
    movf    sGPIO,w            ; turn on indicator LED
    movwf   GPIO               ; update port (copy shadow to GPIO)
    banksel  TMR0
    movf    TMR0,w
    xorlw   .125                ; (125 ticks x 32 us/tick = 4 ms)
    btfss   STATUS,Z
    goto    w_tmr0
```

The additional code doesn't affect the timing of the background task (flashing the LED), because there are only a few additional instructions; they are able to be executed within the 32 μ s available between each "tick" of Timer0.

XC8 implementation

There are no new features to introduce; Timer0 is setup and accessed in the same way as in the last example.

The assembly code above can be implemented with XC8 as:

```
for (;;)
{
    // delay 500 ms while responding to button press
    for (dc = 0; dc < 125; dc++) // repeat 125 times (125 x 4 ms = 500 ms)
    {
        TMR0 = 0; // clear timer0
        while (TMR0 < 125) // repeat for 4 ms (125 x 32 us)
        {
            // check and respond to button press
            sGPIO &= ~(1<<1); // assume button up -> LED off
            if (GP3 == 0) // if button pressed (GP3 low)
                sGPIO |= 1<<1; // turn on LED on GP1
            GPIO = sGPIO; // update port (copy shadow to GPIO)
        }
        // toggle flashing LED
        sGPIO ^= 1<<2; // toggle LED on GP2 using shadow reg
    } // repeat forever
}
```

There is no need to update GPIO after the LED on GP2 is toggled, because GPIO is being continually updated from sGPIO within the inner timer wait loop.

Note the syntax used to set, clear and toggle bits in the shadow GPIO variable, sGPIO:

```
sGPIO |= 1<<1; // turn on LED on GP1
sGPIO &= ~(1<<1); // turn off LED on GP1
sGPIO ^= 1<<2; // toggle LED on GP2
```

We could instead have written:

```
sGPIO |= 0b000010; // turn on LED on GP1
sGPIO &= 0b111101; // turn off LED on GP1
sGPIO ^= 0b000100; // toggle LED on GP2
```

But the left shift ('<<') form more clearly specifies which bit is being operated on.

If we define symbols representing the port bit positions:

```
#define nFLASH 2 // flashing LED on GP2
#define nPRESS 1 // "button pressed" indicator LED on GP1
```

we can write these statements as:

```
sGPIO |= 1<<nPRESS; // turn on indicator LED
sGPIO &= ~(1<<nPRESS); // turn off indicator LED
sGPIO ^= 1<<nFLASH; // toggle flashing LED
```

These symbols can also be used when configuring the port directions:

```
TRISIO = ~(1<<nFLASH|1<<nPRESS); // configure LEDs (only) as outputs
```

This makes the code clearer, more general, and therefore more maintainable.

However, this approach doesn't work well on bigger PICs, which have more than one port. You still need to keep track of which port each pin belongs to, and if you change your pin assignments later, you may well need to make a number of changes throughout your code.

A more robust approach is to make use of *bitfields* within C structures.

For example:

```
struct {
    unsigned    GP0      : 1;
    unsigned    GP1      : 1;
    unsigned    GP2      : 1;
    unsigned    GP3      : 1;
    unsigned    GP4      : 1;
    unsigned    GP5      : 1;
} sGPIObits;
```

It is then possible to refer to each bit as a structure member, for example:

```
sGPIObits.GP1 = 1;
```

and if we also defined a symbol such as:

```
#define sPRESS    sGPIObits.GP1
```

we can then write this as:

```
sPRESS = 1;
```

That's nice – we have “shadow bits” and we can refer to them easily by symbolic names – but there's still a problem. As well as being able to access individual bits, we also need to be able to refer to the whole shadow register as a single variable, to read or update all the bits at once. After all, that's the whole point of using a shadow register.

We want to be able to change a single bit, as in:

```
sGPIObits.GP1 = 1;    // set shadow GP1
```

and also read the whole shadow register in a single operation, as in:

```
GPIO = sGPIO;        // copy shadow register to port
```

How can we do both?

The C *union* construct is intended for exactly this situation, where we need to access the memory holding a variable in more than one way.

We can define for example:

```
union {
    uint8_t      port;                // shadow copy of GPIO
    struct {
        unsigned    GP0      : 1;
        unsigned    GP1      : 1;
        unsigned    GP2      : 1;
        unsigned    GP3      : 1;
        unsigned    GP4      : 1;
        unsigned    GP5      : 1;
    };
} sGPIO;
```

This allows us to refer to the shadow register as `sGPIO.port`, representing the whole port, in a single operation. For example:

```
sGPIO.port = 0;          // clear shadow register

GPIO = sGPIO.port;     // update port (copy shadow to GPIO)
```

We can also refer to the individual shadow bits as, for example:

```
sGPIO.GP1 = 1;         // set shadow GP1
```

If we define symbols representing these shadow bits:

```
#define sFLASH  sGPIO.GP2          // flashing LED (shadow)
#define sPRESS  sGPIO.GP1          // "button pressed" indicator LED (shadow)
```

we can rewrite the previous bit-manipulation statements as:

```
sPRESS = 1;                // turn on indicator LED
sPRESS = 0;                // turn off indicator LED
sFLASH = !sFLASH;         // toggle flashing LED
```

and, very concisely:

```
sPRESS = !BUTTON;         // turn on indicator only if button pressed
```

Besides clarity and conciseness, a big advantage of this technique is that, if (on a larger PIC) you were to move one of these functions (such as the flashing LED) to another port, you only need to modify the symbol definition and perhaps your initialisation routine. The rest of your program could stay the same – these statements would still work.

Defining the shadow register as a union incorporating a bitfield structure may seem like a lot of trouble for an apparently small benefit, but it's an elegant approach that will pay off as your applications become more complex.

Complete program

Here is how this shadow register union / bitfield structure definition is used in practice:

```

/*****
*   Description:      Lesson 2, example 2b
*
*   Demonstrates use of Timer0 to maintain timing of background actions
*   while performing other actions in response to changing inputs
*
*   One LED simply flashes at 1 Hz (50% duty cycle).
*   The other LED is only lit when the pushbutton is pressed
*
*   Uses union / bitfield structure to represent shadow register
*
*****/
*   Pin assignments:
*   GP1 = "button pressed" indicator LED
*   GP2 = flashing LED
*   GP3 = pushbutton switch (active low)
*****/

#include <xc.h>
#include <stdint.h>

```

```

/***** CONFIGURATION *****/
// int reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sFLASH  sGPIO.GP2      // flashing LED (shadow)
#define sPRESS  sGPIO.GP1      // "button pressed" indicator LED (shadow)
#define BUTTON  GPIObits.GP3    // pushbutton

/***** GLOBAL VARIABLES *****/
union {
    uint8_t      port;          // shadow copy of GPIO
    struct {
        unsigned  GP0      : 1;
        unsigned  GP1      : 1;
        unsigned  GP2      : 1;
        unsigned  GP3      : 1;
        unsigned  GP4      : 1;
        unsigned  GP5      : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    uint8_t  dc;                // delay counter

    //*** Initialisation

    // configure port
    GPIO = 0;                   // start with all LEDs off
    sGPIO.port = 0;             // update shadow
    TRISIO = ~(1<<1|1<<2);     // configure GP1 and GP2 (only) as outputs

    // configure Timer0
    OPTION_REGbits.T0CS = 0;    // select timer mode
    OPTION_REGbits.PSA = 0;     // assign prescaler to Timer0
    OPTION_REGbits.PS = 0b100;  // prescale = 32
                                // -> increment every 32 us

    //*** Main loop
    for (;;)
    {
        // delay 500 ms while responding to button press
        for (dc = 0; dc < 125; dc++) // repeat 125 times (125 x 4 ms = 500 ms)
        {
            TMR0 = 0;           // clear timer0
            while (TMR0 < 125) // repeat for 4 ms (125 x 32 us)
            {
                sPRESS = !BUTTON; // turn on LED only if button pressed
                GPIO = sGPIO.port; // update port (copy shadow to GPIO)
            }
        }
        // toggle flashing LED
        sFLASH = !sFLASH;      // toggle flashing LED (shadow)

    } // repeat forever
}

```


Comparisons

Here is the resource usage summary for the “Flash an LED while responding to a pushbutton” programs (the C version defining the shadow register as a union containing a bitfield structure, as above):

Flash+PB_LED

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	44	37	38	31	2	2
XC8 (Free mode)	31	28	78	72	3	2

The C source code is comparatively long in this example, because of the shadow register union / bitfield structure definition. It’s a big part of the source code – something you wouldn’t normally bother with, for such a small program. But we’ll keep doing it this way, because it’s good practice that will serve us well as our programs become longer, and the extra lines of variable definition won’t seem like such a big deal.

Example 3: Switch debouncing

The [previous lesson](#) demonstrated one method commonly used to debounce switches: sampling the switch state periodically, and only considering it to have definitely changed when it has been in the new state for some minimum number of successive samples.

This “counting algorithm” was given as:

```
count = 0
while count < max_samples
    delay sample_time
    if input = required_state
        count = count + 1
    else
        count = 0
end
```

As explained in [mid-range lesson 4](#), this can be simplified by using a timer, since the timer increments automatically:

```
reset timer
while timer < debounce_time
    if input ≠ required_state
        reset timer
end
```

This algorithm was implemented in assembler, to wait for and debounce a “button down” event, as follows:

```
wait_dn banksel TMR0
chk_dn  clrf    TMR0           ; reset timer
        btfsc  GPIO,GP3      ; check for button press (GP3 low)
        goto   wait_dn       ; continue to reset timer until button down
        movf   TMR0,w         ; has 10 ms debounce time elapsed?
        xorlw  .157           ; (157 = 10ms/64us)
        btfss  STATUS,Z       ; if not, continue checking button
        goto   chk_dn
```

This code assumes that Timer0 is available, and is in timer mode, with a 1 MHz instruction clock and a 1:64 prescaler, giving 64 μ s per tick.

Although mid-range PICs have more than one timer, they remain a scarce resource, and it is likely that Timer0 is being used for something else, and so is not available for switch debouncing. As we'll see in the [next lesson](#), it can be better to use a regular timer-driven interrupt for switch debouncing, allowing a single timer (driving the interrupt) to be used for a number of tasks.

But if you're not using Timer0 for anything else, using it for switch debouncing is perfectly reasonable.

This technique was demonstrated by applying this timer-based debouncing method to the "toggle an LED on pushbutton press" program developed in [mid-range assembler lesson 3](#).

XC8 implementation

Timer0 can be configured for timer mode, with a 1:64 prescaler, by:

```
OPTION_REGbits.T0CS = 0;           // select timer mode
OPTION_REGbits.PSA = 0;           // assign prescaler to Timer0
OPTION_REGbits.PS = 0b101;       // prescale = 64
                                   // -> increment every 64 us
```

This is the same as for the 1:32 prescaler examples, above, except that the PS<2:0> bits are set to '101' instead of '100'.

The timer-based debounce algorithm, given above in pseudo-code, is readily translated into C:

```
TMR0 = 0;                          // reset timer
while (TMR0 < 157)                 // wait at least 10 ms (157 x 64 us = 10 ms)
    if (GPIObits.GP3 == 1)         // if button up,
        TMR0 = 0;                  // restart wait
```

Using C macros

This fragment of code is one that we might want to use a number of times, perhaps modified to debounce switches on inputs other than GP3, in this or other programs.

As we saw in [mid-range lesson 5](#), the MPASM assembler provides a *macro* facility, which allows a parameterised segment of code to be defined once and then inserted multiple times into the source code.

Macros can also be used when programming in C.

For example, we could define our debounce routine as a macro as follows:

```
#define DEBOUNCE 10*1000/256        // switch debounce count = 10 ms/(256us/tick)

// DbnceLo()
//
// Debounce switch on given input pin
// Waits for switch input to be low continuously for DEBOUNCE*256/1000 ms
//
// Uses: TMR0           Assumes: TMR0 running at 256 us/tick
//
#define DbnceLo(PIN) TMR0 = 0;      /* reset timer           */ \
    while (TMR0 < DEBOUNCE)        /* wait until debounce time */ \
        if (PIN == 1)              /* if input high,         */ \
            TMR0 = 0;              /* restart wait           */ \
```

Note that a backslash (‘\’) is placed at the end of all but the last line, to continue the macro definition over multiple lines. To make the backslashes visible to the C pre-processor, the older “/* */” style comments must be used, instead of the newer “//” style.

This macro can then be used within your program as, for example:

```
DbnceLo (GPIObits.GP3);    // wait until button pressed (GP3 low)
```

You can define macros toward the start of your source code, but as you build your own library of useful macros, you would normally keep them together in one or more header files, such as “stdmacros.h”, and reference them from your main program, using the #include directive.

Complete program

Here is how this timer-based debounce code (without using macros) fits into the XC8 version of the “toggle an LED on pushbutton press” program:

```

/*****
*   Description:      Lesson 2, example 3a
*
*   Demonstrates use of Timer0 to implement debounce counting algorithm
*
*   Toggles LED when pushbutton is pressed then released
*
*****/
*   Pin assignments:
*       GP1 = flashing LED
*       GP3 = pushbutton switch (active low)
*
*****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// int reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sFLASH    sGPIO.GP1           // flashing LED (shadow)
#define BUTTON    GPIObits.GP3       // pushbutton

/***** GLOBAL VARIABLES *****/
union {
    uint8_t        port;              // shadow copy of GPIO
    struct {
        unsigned   GP0      : 1;
        unsigned   GP1      : 1;
        unsigned   GP2      : 1;
        unsigned   GP3      : 1;
        unsigned   GP4      : 1;
        unsigned   GP5      : 1;
    };
} sGPIO;

```

```

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    GPIO = 0;                // start with LED off
    sGPIO.port = 0;         // update shadow
    TRISIO = 0b111101;     // configure GP1 (only) as an output

    // configure Timer0
    OPTION_REGbits.T0CS = 0; // select timer mode
    OPTION_REGbits.PSA = 0;  // assign prescaler to Timer0
    OPTION_REGbits.PS = 0b101; // prescale = 64
                                // -> increment every 64 us

    /*** Main loop
    for (;;)
    {
        // wait for button press, debounce using timer0:
        TMR0 = 0;            // reset timer
        while (TMR0 < 157) // wait at least 10 ms (157 x 64 us = 10 ms)
            if (BUTTON == 1) // if button up,
                TMR0 = 0;    // restart wait

        // toggle LED
        sFLASH = !sFLASH;   // toggle flashing LED (shadow)
        GPIO = sGPIO.port;  // write to GPIO

        // wait for button release, debounce using timer0:
        TMR0 = 0;            // reset timer
        while (TMR0 < 157) // wait at least 10ms (157 x 64us = 10 ms)
            if (BUTTON == 0) // if button down,
                TMR0 = 0;    // restart wait

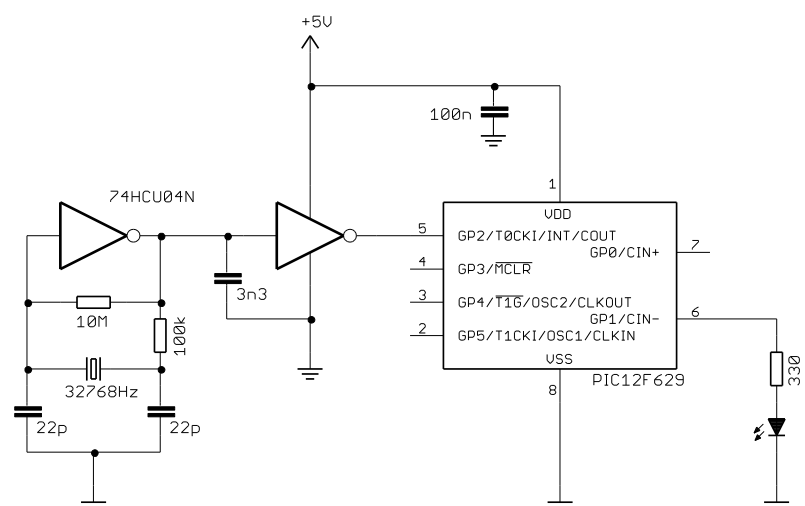
    } // repeat forever
}

```

Example 4: Using Counter Mode

Until now we've used Timer0 in "timer mode", where it is clocked by the PIC's instruction clock, which runs at one quarter the speed of the processor clock (i.e. 1 MHz when the 4 MHz internal RC oscillator is used). As we saw in [mid-range lesson 4](#), the timer can instead be used in "counter mode", where it counts transitions (rising or falling) on the PIC's T0CKI input.

We can use the example from that lesson to show how Timer0 can be used as a counter, using C: Timer0 is driven by an external 32.768 kHz crystal oscillator (as shown on the right), providing a time base that can be used to flash an LED at a more accurate 1 Hz.



To configure the [Gooligum training board](#) for this example, close jumper JP22 to connect the 32 kHz clock signal to T0CKI, and close jumpers JP3 and JP12 to enable the external $\overline{\text{MCLR}}$ pull-up resistor (not shown in this diagram, for clarity) and the LED on GP1. If you are using Microchip's Low Pin Count Demo Board, you will need to build the oscillator circuit separately, as described in [baseline assembler lesson 5](#).

If the 32.768 kHz clock input is divided (prescaled) by 128, bit 7 of TMR0 will cycle at 1 Hz.

To configure Timer0 for counter mode (external clock on T0CKI) with a 1:128 prescale ratio, we need to set the T0CS bit to '1', PSA to '0' and PS<2:0> to '110'.

This was done in [mid-range assembler lesson 4](#) by:

```

movlw    b'11110110'    ; configure Timer0:
          ; --1-----    counter mode (T0CS = 1)
          ; ----0---    prescaler assigned to Timer0 (PSA = 0)
          ; -----110    prescale = 128 (PS = 110)
banksel  OPTION_REG    ; -> increment at 256 Hz with 32.768 kHz input
movwf   OPTION_REG

```

The value of T0SE bit is irrelevant; we don't care if the counter increments on the rising or falling edge of the input clock signal – only the frequency is important. Either edge will do.

Bit 7 of TMR0 (which is cycling at 1 Hz) was then continually copied to GP1 (using a shadow register), as follows:

```

loop     ; transfer TMR0<7> to GP1
        clrfsf    sGPIO          ; assume TMR0<7>=0 -> LED off
        banksel  TMR0
        btfsf    TMR0,7          ; if TMR0<7>=1
        bsf     sGPIO,GP1        ; turn on LED

        movf     sGPIO,w         ; copy shadow to GPIO
        banksel  GPIO
        movwf   GPIO

        ; repeat forever
        goto    loop

```

XC8 implementation

Configuring Timer0 is simply:

```

OPTION_REGbits.T0CS = 1;    // select counter mode
OPTION_REGbits.PSA = 0;    // assign prescaler to Timer0
OPTION_REGbits.PS = 0b110; // prescale = 128
                          // -> incr at 256 Hz with 32.768 kHz input

```

To copy bit 7 of TMR0 to the LED (via a shadow bit), we can use the following construct:

```

sFLASH = 0;                // assume TMR<7>=0 -> LED off
if (TMR0 & 1<<7)           // if TMR0<7>=1
    sFLASH = 1;            // turn on LED

```

This works because the expression "1<<7" equals 10000000 binary, so the result of ANDing TMR0 with 1<<7 will only be non-zero if TMR0<7> is set.

Or we could write this equivalently as:

```

sFLASH = (TMR0 & 1<<7) != 0;    // sFLASH = TMR0<7>

```

Which construct you use is, as ever, a matter of personal style; we'll use the second version here.

Complete program

Here is the XC8 version of the “flash an LED using crystal-driven timer” program:

```

/*****
*   Description:      Lesson 2, example 4
*
*   Demonstrates use of Timer0 in counter mode
*
*   LED flashes at 1 Hz (50% duty cycle),
*   with timing derived from 32.768 kHz input on T0CKI
*
*****/
*   Pin assignments:
*       GP1   = flashing LED
*       T0CKI = 32.768 kHz signal
*
*****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sFLASH  sGPIO.GP1           // flashing LED (shadow)

/***** GLOBAL VARIABLES *****/
union {
    uint8_t      port;           // shadow copy of GPIO
    struct {
        unsigned GP0    : 1;
        unsigned GP1    : 1;
        unsigned GP2    : 1;
        unsigned GP3    : 1;
        unsigned GP4    : 1;
        unsigned GP5    : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    TRISIO = ~(1<<1);           // configure GP1 (only) as an output

    // configure Timer0
    OPTION_REGbits.T0CS = 1;     // select counter mode
    OPTION_REGbits.PSA = 0;      // assign prescaler to Timer0
    OPTION_REGbits.PS = 0b110;   // prescale = 128
    // -> incr at 256 Hz with 32.768 kHz input

```

```
//*** Main loop
for (;;)
{
    // TMR0<7> cycles at 1 Hz, so continually copy to LED
    sFLASH = (TMR0 & 1<<7) != 0;    // sFLASH = TMR0<7>

    GPIO = sGPIO.port;            // copy shadow to GPIO
} // repeat forever
}
```

Summary

These examples have demonstrated that Timer0 can be effectively configured and accessed using the XC8 compiler, with the program algorithms being able to be expressed quite succinctly in C.

We've also seen that using symbolic names and macros can help make your code more maintainable, and how the union and bitfield structure constructs can be used to make it possible to access both a whole variable and its individual bits, in an elegant way.

In the [next lesson](#) we'll see how interrupts can be implemented using XC8.

Introduction to PIC Programming

Programming Mid-Range PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 3: Introduction to Interrupts

As we saw in [mid-range lesson 6](#), the *interrupt* facility available on mid-range PICs is especially useful, making it much easier to implement regular “background” tasks (such as refreshing a multiplexed display – see for example [baseline C lesson 5](#)) and allow programs to respond in a timely manner to external events, without having to sit in a *busy-wait*, or polling loop. Both of these applications of interrupts are demonstrated in this lesson.

This lesson revisits the material from [mid-range lesson 6](#), introducing external and timer interrupts (driven by Timer0) and some of their applications, such as running background tasks and switch debouncing.

As usual, the examples are re-implemented using Microchip’s XC8 compiler¹ (running in “Free mode”), introduced in [lesson 1](#).

In summary, this lesson covers:

- Introduction to interrupts on the mid-range PIC architecture
- Interrupt handling, using XC8
- Timer-driven interrupts
- Debouncing single switches with timer-driven interrupts
- External interrupts on the INT pin

Note that this tutorial series assumes a working knowledge of the C language; it does **not** attempt to teach C.

Interrupts

An *interrupt* is a means of interrupting the main program flow in response to an event, so that the event can be handled, or *serviced*. The event (referred to an interrupt *source*) can be internal to the PIC, such as a timer overflowing, or external, such as a change on an input pin.

When the interrupt is triggered, program execution immediately jumps to an *interrupt service routine (ISR)*, which, in the mid-range PIC architecture, is always located at address 0004h (the “interrupt vector”).

The XC8 compiler hides this detail; if a function is defined with the qualifier ‘*interrupt*’, the compiler considers it to be an interrupt service routine, and places it at the correct address, automatically. Of course, this means that, on mid-range PICs, the ‘*interrupt*’ qualifier can only be used with one function, as there is only one interrupt vector in the mid-range architecture.

¹ Available as a free download from www.microchip.com.

The ISR must save the current processor state, or *context* (i.e. the contents of any registers which the ISR will modify, such as *W* and *STATUS*), service the interrupt, and then restore the context before returning to the main program. In this way, the main program will never “notice” that the interrupt has happened – the interrupt will be completely transparent, except for whatever action the interrupt service routine was intended to perform.

Again, the XC8 compiler takes care of this implementation detail, automatically adding appropriate context save and restore code to the ‘*interrupt*’ function.

Timer0 Interrupts

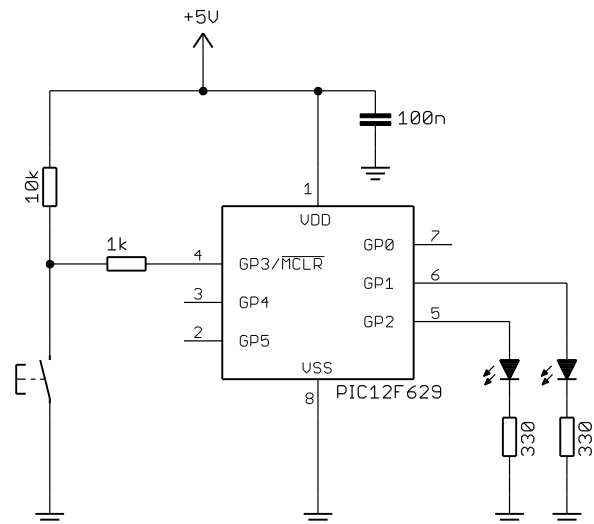
Timer0 can be used to regularly generate interrupts, which can drive “background” tasks, such as:

- Generating a regular output; for example flashing an LED.
- Monitoring and debouncing inputs

Meanwhile, a “main program” can continue to perform other “foreground” tasks.

The examples in this section illustrate these techniques, using the circuit from [lesson 2](#), shown on the right.

If you have the [Gooligum training board](#), close jumpers JP3, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP1 and GP2.



Example 1a: Flashing an LED

To begin, we’ll simply flash an LED, without attempting to make it flash at exactly 1 Hz.

We saw in [mid-range lesson 4](#) that, given a 1 MHz instruction clock with maximum prescaling (1:256), the longest period that Timer0 can generate is $256 \times 256 \times 1 \mu\text{s} = 65.5 \text{ ms}$. Therefore, if we configured the PIC to use a 4 MHz clock, and set up Timer0 in timer mode with a 1:256 prescaler, TMR0 would *overflow* (rollover from 255 to 0) every 65.5 ms.

If we then enabled Timer0 interrupts, the interrupt would be triggered on every TMR0 overflow, i.e. every 65.5 ms. So the interrupt service routine (ISR) would be called every 65.5 ms.

If the ISR toggled an LED every time it was called, the LED would change state every 65.5 ms – it would flash with a period of $65.5 \text{ ms} \times 2 = 131 \text{ ms}$, giving a frequency of 7.6 Hz.

Having an LED flash as 7.6 Hz is not ideal, but the flashing is visible (just), and that’s the slowest flash rate we can generate with the simple approach described above. So we’ll start there.

The assembler code in [mid-range lesson 6](#) configured the port and Timer0, before enabling the Timer0 interrupt by setting the TOIE (Timer0 interrupt enable) and GIE (global interrupt enable) bits in the INTCON register:

```
; enable interrupts
movlw 1<<GIE|1<<TOIE ; enable Timer0 and global interrupts
movwf INTCON
```

The interrupt service routine began by saving the processor context, and then reset, or cleared, the Timer0 interrupt flag (TOIF) to show that this Timer0 overflow event has been handled – if this is not done, the interrupt would immediately re-trigger, as soon as the ISR has exited.

The interrupt service routine then toggled the LED, indirectly, by toggling the bit corresponding to the LED in a shadow register:

```
movf    sGPIO,w           ; only update shadow register
xorlw   1<<nLED
movwf   sGPIO
```

This was done to avoid potential read-modify-write problems (described in [baseline assembler lesson 2](#)).

Finally, the ISR restored the processor context, before exiting and returning control to the main program.

The body of the main program then had only a single task to perform – to repeatedly copy the contents of the shadow register to the GPIO port, to make the changes made within the ISR visible (literally!):

```
main_loop
    ; continually copy shadow GPIO to port
    movf    sGPIO,w
    banksel GPIO
    movwf   GPIO

    ; repeat forever
    goto   main_loop
```

XC8 implementation

As mentioned above, the XC8 compilers hide much of the complexity associated with handling interrupts, such as saving and restoring the processor context.

The interrupt service routine is implemented as a function, defined with the qualifier ‘interrupt’.

For example:

```
void interrupt isr(void)
```

Note that the interrupt function should be declared as type void, and must not take any parameters, because it is never explicitly called from anywhere – nothing is passed to it, and nothing is returned. It just “happens”, whenever an interrupt is triggered. The name of the interrupt function is not important; you don’t have to call it ‘isr’.

Since direct parameter passing isn’t possible, any data passed between the ISR and the main program must be held in global variables (declared outside any function), so that both the interrupt function and main() (and any other functions) can access them.

Additionally, any global variable which may be modified by the ISR must be declared as ‘volatile’, to warn the compiler from eliminating apparently redundant references to those variables in the main program.

We’ll continue to use the union construct introduced in the previous lesson for the shadow copy of GPIO. Since it is accessed by both the ISR and the main program, it must be declared as a global variable, before main() or the interrupt function, and qualified as ‘volatile’:

```
/***** GLOBAL VARIABLES *****/
volatile union {                               // shadow copy of GPIO
    uint8_t      port;
    struct {
        unsigned GP0      : 1;
        unsigned GP1      : 1;
        unsigned GP2      : 1;
        unsigned GP3      : 1;
        unsigned GP4      : 1;
        unsigned GP5      : 1;
    };
} sGPIO;
```

Since we don't need to worry about saving or restoring the processor context, the XC8 version of the ISR can be very simple:

```
void interrupt isr(void)
{
    /*** Service Timer0 interrupt
    //
    // TMR0 overflows every 65.5 ms
    //
    // Flashes LED at ~7.6 Hz by toggling on each interrupt
    // (every ~65.5 ms)
    //
    // (only Timer0 interrupts are enabled)
    //
    INTCONbits.TOIF = 0;           // clear interrupt flag

    // toggle LED
    sF_LED = ~sF_LED;           // (via shadow register)
}
```

The symbol 'sF_LED' had been defined previously, to help make the code more maintainable:

```
// Pin assignments
#define sF_LED  sGPIO.GP2           // flashing LED (shadow)
```

In the main program, we configure the port and Timer0, as we have done before:

```
// configure port
GPIO = 0;           // start with all LEDs off
sGPIO.port = 0;    // update shadow
TRISIO = ~(1<<2); // configure GP2 (only) as an output

// configure Timer0
OPTION_REGbits.T0CS = 0; // select timer mode
OPTION_REGbits.PSA = 0; // assign prescaler to Timer0
OPTION_REGbits.PS = 0b111; // prescale = 256
// -> increment every 256 us
```

Having configured the port and timer, we're ready to enable the Timer0 interrupt, which, as we saw above, is done by setting the TOIE and GIE bits in the INTCON register.

This could be done by:

```
// enable interrupts
INTCONbits.TOIE = 1; // enable Timer0 interrupt
INTCONbits.GIE = 1; // enable global interrupts
```

However, XC8 defines a macro, 'ei()', which is intended to be used to enable interrupts globally, and is equivalent to 'INTCONbits.GIE = 1'.

Similarly, there is a 'di()' macro, used to disable all interrupts, equivalent to 'INTCONbits.GIE = 0'.

So, in keeping with the XC8 conventions, the Timer0 interrupt should be enabled by:

```
// enable interrupts
INTCONbits.TOIE = 1; // enable Timer0 interrupt
ei();                // enable global interrupts
```

Finally, we need to continually copy the shadow register to GPIO, which can be done by:

```

/***/ Main loop
for (;;)
{
    // continually copy shadow GPIO to port
    GPIO = sGPIO.port;

} // repeat forever

```

Complete program

Here is how these code fragments fit together:

```

/*****
*
* Description: Lesson 3, example 1a
*
* Demonstrates use of Timer0 interrupt to perform a background task
*
* Flash LED at approx 7.6 Hz (50% duty cycle)
*
*****/
*
* Pin assignments:
* GP2 = flashing LED
*
*****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sF_LED sGPIO.GP2 // flashing LED (shadow)

/***** GLOBAL VARIABLES *****/
volatile union { // shadow copy of GPIO
    uint8_t port;
    struct {
        unsigned GP0 : 1;
        unsigned GP1 : 1;
        unsigned GP2 : 1;
        unsigned GP3 : 1;
        unsigned GP4 : 1;
        unsigned GP5 : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    /***/ Initialisation

```

```

// configure port
GPIO = 0; // start with all LEDs off
sGPIO.port = 0; // update shadow
TRISIO = ~(1<<2); // configure GP2 (only) as an output

// configure Timer0
OPTION_REGbits.T0CS = 0; // select timer mode
OPTION_REGbits.PSA = 0; // assign prescaler to Timer0
OPTION_REGbits.PS = 0b111; // prescale = 256
// -> increment every 256 us

// enable interrupts
INTCONbits.T0IE = 1; // enable Timer0 interrupt
ei(); // enable global interrupts

//*** Main loop
for (;;)
{
    // continually copy shadow GPIO to port
    GPIO = sGPIO.port;

} // repeat forever
}

/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt_isr(void)
{
    //*** Service Timer0 interrupt
    //
    // TMR0 overflows every 65.5 ms
    //
    // Flashes LED at ~7.6 Hz by toggling on each interrupt
    // (every ~65.5 ms)
    //
    // (only Timer0 interrupts are enabled)
    //
    INTCONbits.T0IF = 0; // clear interrupt flag

    // toggle LED
    sF_LED = ~sF_LED; // (via shadow register)
}

```

Example 1b: Slower flashing

The LED in the last example flashed at around 7.6 Hz. Since the longest possible interval between Timer0 interrupts is 65.5 ms (with a 4 MHz processor clock), to flash the LED any slower, we can't toggle it on every interrupt; we have to skip some of them. That means counting each interrupt, and only toggling the LED when the count reaches a certain value.

A simple way to implement this, if we are not concerned with exact timing, is to use an 8-bit counter, and to let it reach 255 before toggling the LED when it overflows to 0.

If, every time an interrupt is triggered by a Timer0 overflow, the ISR increments a counter, we're essentially implementing a 16-bit timer, based on Timer0, with TMR0 as the least significant eight bits, and the counter incremented by the ISR being the most significant eight bits.

If the ISR increments the counter whenever Timer0 overflows (every 256 *ticks* of TMR0), and it toggles the LED whenever the counter overflows (every 256 interrupts), the LED is being toggled every $N \times 256 \times 256$ (where N is the prescale ratio) instruction cycles.

Assuming a 1 MHz instruction clock, LED will be toggled every $N \times 256 \times 256 \mu\text{s} = N \times 65.536 \text{ ms}$.

We can make the LED flash at close to 1 Hz by choosing $N = 8$ (prescale ratio of 1:8). The resulting toggle period is $8 \times 256 \times 256 \mu\text{s} = 524.3 \text{ ms}$, giving a flash rate of 0.95 Hz – close enough!

XC8 implementation

To implement the Timer0 overflow counter, we'll need a variable to store it in.

Since this variable only needs to be used by the interrupt service routine, to be consistent with good modular programming practice, we should make it private to (defined within) the interrupt function:

```
void interrupt isr(void)
{
    static uint8_t cnt_t0 = 0;        // counts timer0 overflows

    // (body of ISR goes here)
}
```

Note that this variable is declared as being 'static'; this is very important. The counter must retain its value between interrupts, so that it can be incremented by successive interrupts. To ensure that the counter continues to exist, preserving its value, outside the interrupt function, it must be declared as 'static'.

Note also that if the counter variable is initialised, as part of its definition. You might think that, because the definition is within the interrupt function, that this initialisation (clearing the counter) will happen every time an interrupt occurs, losing the value of the counter. But no – all static variables are initialised only once, by the start-up code generated by the C compiler, before the `main()` function starts executing.

We then need to add instructions to the ISR to increment this counter, and toggle the LED only when it overflows back to zero:

```
// toggle LED every 256 interrupts (524 ms)
++cnt_t0;                // increment interrupt count (every 2.048 ms)
if (cnt_t0 == 0)        // if count overflow (every 256 interrupts),
    sF_LED = ~sF_LED;   // toggle LED (via shadow register)
```

This could have been written more succinctly as:

```
// toggle LED every 256 interrupts (524 ms)
if (++cnt_t0 == 0)      // increment count; if overflow (every 524 ms),
    sF_LED = ~sF_LED;   // toggle LED (via shadow register)
```

Whether you choose to sacrifice readability to save a line of source code is a question of personal style.

Here is the complete ISR, with these changes:

```
void interrupt isr(void)
{
    static uint8_t cnt_t0 = 0;        // counts timer0 overflows

    /*** Service Timer0 interrupt
    //
    // TMR0 overflows every 2.048 ms
    //
    // Flashes LED at ~0.95 Hz by toggling on every 256th interrupt
    // (every ~524 ms)
```

```

// (only Timer0 interrupts are enabled)
//
INTCONbits.T0IF = 0;           // clear interrupt flag

// toggle LED every 256 interrupts (524 ms)
++cnt_t0;                     // increment interrupt count (every 2.048 ms)
if (cnt_t0 == 0)              // if count overflow (every 256 interrupts),
    sF_LED = ~sF_LED;        // toggle LED (via shadow register)
}

```

And finally the configuration of Timer0 needs to be changed, to select a 1:8 prescaler:

```

// configure Timer0
OPTION_REGbits.T0CS = 0;      // select timer mode
OPTION_REGbits.PSA = 0;      // assign prescaler to Timer0
OPTION_REGbits.PS = 0b010;   // prescale = 8
// -> increment every 8 us

```

With these changes to the code in the first example, the LED will flash at a much more sedate 0.95 Hz.

Example 1c: Flashing an LED at exactly 1 Hz

What if we needed (for some reason) to flash the LED at exactly 1 Hz, given an accurate 4 MHz processor clock? As discussed in detail in [mid-range lesson 6](#), there are a number of pitfalls inherent in trying to use Timer0 to generate a cycle-exact time base.

But as we saw, these problems can be overcome, relatively easily.

To use Timer0 to provide a precise time base to drive an interrupt:

- Do not use the prescaler (assign it to the watchdog timer).
- Do not load a fixed start value into the timer.

Instead, add an offset to the current timer value, making the timer “skip forward” by an appropriate amount, shortening the timer cycle from 256 counts to whatever period you require.

- Adjust the offset to allow for the fact that the timer is inhibited for two cycles after it is written, and that the timer increments once (if no prescaler is used) during the add instruction.

This means that the offset to be added must be 3 cycles larger than you may expect, to achieve a given timer period.

In the example in [mid-range lesson 6](#), we used the following assembler code:

```

movlw    .256-.250+.3    ; add value to Timer0
banksel  TMR0           ; for overflow after 250 counts
addwf    TMR0,f

```

to make Timer0 overflow after 250 cycles, instead of the usual 256 cycles (with no prescaler). This was done after every Timer0 overflow (i.e. within the interrupt service routine), so that the interrupt is triggered precisely every 250 instruction cycles (every 250 μ s, given a 4 MHz processor clock).

Toggling the LED every 500 ms means toggling after every $500 \text{ ms} \div 250 \mu\text{s} = 2000$ interrupts.

This means that the ISR must be able to count to 2000, so that it can toggle the LED after 2000 interrupts.

In the assembler version, this was realised by using two 8-bit variables, one counting interrupts to create a 10 ms time base, the other counting these 10 ms intervals to generate the 500 ms period we need.

But since we're using C here, we may as well take advantage of its ability to easily work with larger quantities, and simply use a single 16-bit variable to count interrupts.

XC8 implementation

Since the timer overflow counter is only accessed by the interrupt service routine, it should be defined within the interrupt function, as was done in the last example:

```
void interrupt isr(void)
{
    static uint16_t cnt_t0 = 0;    // counts timer0 overflows

    // (body of ISR goes here)
}
```

Note again that, because this variable needs to be able to count up to 2000, it is defined as a 16-bit integer (`uint16_t`), instead of the 8-bit type (`uint8_t`) we used in the previous example.

To make the Timer0 interrupt occur every 250 cycles, instead of the usual 256, we need to add an appropriate offset to `TMR0`, within the ISR, as follows:

```
TMR0 += 256-250+3;           // add value to Timer0
                               // for overflow after 250 counts
```

It is then a simple matter to count interrupts and toggle the LED after 500 ms (2000 counts):

```
// toggle LED every 500 ms
++cnt_t0;           // increment interrupt count (every 250 us)
if (cnt_t0 == 500000/250) { // if count overflow (every 500 ms),
    cnt_t0 = 0;       // reset count
    sF_LED = ~sF_LED; // toggle LED (via shadow register)
```

Finally, in the initialisation part of the main program, we need to configure Timer0 with no prescaler:

```
// configure Timer0
OPTION_REGbits.T0CS = 0; // select timer mode
OPTION_REGbits.PSA = 1;  // no prescaler (assigned to WDT)
                          // -> increment every 1 us
```

With these modifications in place, the LED will now flash with a frequency of exactly 1 Hz, assuming that the processor clock is exactly 4 MHz (which, since we are using the internal RC oscillator, it will not be the case; it's not that accurate. Nevertheless, the LED flashes every 4,000,000 processor cycles, precisely).

Complete program

Here is how the code fragments above fit together:

```
/******
 *
 * Description: Lesson 3, example 1c
 *
 * Demonstrates use of Timer0 interrupt to perform a background task
 *
 * Flash LED at exactly 1 Hz (50% duty cycle)
 *
 *****/
 *
 * Pin assignments:
 * GP2 = flashing LED
 *
 *****/
```



```

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sF_LED  sGPIO.GP2           // flashing LED (shadow)

/***** GLOBAL VARIABLES *****/
volatile union {                   // shadow copy of GPIO
    uint8_t      port;
    struct {
        unsigned GP0    : 1;
        unsigned GP1    : 1;
        unsigned GP2    : 1;
        unsigned GP3    : 1;
        unsigned GP4    : 1;
        unsigned GP5    : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    GPIO = 0;           // start with all LEDs off
    sGPIO.port = 0;    // update shadow
    TRISIO = ~(1<<2); // configure GP2 (only) as an output

    // configure Timer0
    OPTION_REGbits.T0CS = 0; // select timer mode
    OPTION_REGbits.PSA = 1;  // no prescaler (assigned to WDT)
                             // -> increment every 1 us

    // enable interrupts
    INTCONbits.T0IE = 1;    // enable Timer0 interrupt
    ei();                   // enable global interrupts

    /*** Main loop
    for (;;)
    {
        // continually copy shadow GPIO to port
        GPIO = sGPIO.port;

    } // repeat forever
}

/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{

```

```

static uint16_t cnt_t0 = 0;    // counts timer0 overflows

/** Service Timer0 interrupt
//
// TMR0 overflows every 250 clocks = 250 us
// Flashes LED at 1 Hz by toggling on every 2000th interrupt
// (every 500 ms)
//
// (only Timer0 interrupts are enabled)
//
TMR0 += 256-250+3;           // add value to Timer0
// for overflow after 250 counts
INTCONbits.T0IF = 0;       // clear interrupt flag

// toggle LED every 500 ms
++cnt_t0;                   // increment interrupt count (every 250 us)
if (cnt_t0 == 500000/250) { // if count overflow (every 500 ms),
    cnt_t0 = 0;             // reset count
    sF_LED = ~sF_LED;      // toggle LED (via shadow register)
}
}

```

Comparisons

As we've done before, we can compare the length of the source code (ignoring comments and white space) versus program and data memory utilisation for this XC8 version with the corresponding assembly version (from mid-range lesson 6), to illustrate any trade-offs between programmer efficiency and resource-usage efficiency. Longer source code implies more time spent by the programmer writing the code, and more time spent debugging or maintaining the code. Understanding these trade-offs, and the relative value of your time versus device cost (having less efficient code means that you may need a bigger, more expensive, device to hold it), is key to whether you choose to develop in C or assembler:

Flash_LED-50p-int-1Hz

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	64	49	5
XC8 (Free mode)	32	85	8

Once again, the C source code is less than half as long as the assembler source, while the code generated by XC8 (with optimisation disabled) is significantly larger than the hand-written assembly version.

Example 2: Flash LED while responding to input

Now that we have a timer-driven interrupt flashing the LED on GP2 at 1 Hz, that flashing will continue, “on its own”, independently of whatever the main program code is doing. This is the main reason for using a timer interrupt to drive a background process like this; once the process is set up, you do not need to worry about maintaining it in the main code. It may seem complex to set up the interrupt code, but, once done, it makes your main code much easier to write.

To illustrate this, we can re-implement example 2 from [lesson 2](#), where we the LED on GP1 is lit whenever the pushbutton is pressed, while the LED on GP2 continues to flash steadily at 1 Hz.

XC8 implementation

[Lesson 2](#) included this piece of code to light the LED on GP1 only when the pushbutton on GP3 is pressed:

```
sGPIO &= ~(1<<1);           // assume button up -> LED off
if (GP3 == 0)               // if button pressed (GP3 low)
    sGPIO |= 1<<1;         // turn on LED on GP1
GPIO = sGPIO;               // update port (copy shadow to GPIO)
```

If we declare our sGPIO union as in example 1, and define symbols to represent the pins:

```
#define sB_LED  sGPIO.GP1           // "button pressed" indicator LED (shadow)
#define sF_LED  sGPIO.GP2           // flashing LED (shadow)
#define BUTTON  GPIObits.GP3       // pushbutton
```

We can rewrite this as:

```
sB_LED = 0;                  // assume button up -> indicator LED off
if (BUTTON == 0)            // if button pressed (low)
    sB_LED = 1;             // turn on indicator LED
GPIO = sGPIO.port;         // update port (copy shadow to GPIO)
```

In the main loop in example 1, above, we are doing nothing but copying the shadow register to GPIO:

```
for (;;)
{
    // continually copy shadow GPIO to port
    GPIO = sGPIO.port;

} // repeat forever
```

All we need do, then, is to insert the pushbutton-handling code into the main loop:

```
for (;;)
{
    // check and respond to button press
    sB_LED = 0;                // assume button up -> indicator LED off
    if (BUTTON == 0)           // if button pressed (low)
        sB_LED = 1;           // turn on indicator LED

    // continually copy shadow GPIO to port
    GPIO = sGPIO.port;

} // repeat forever
```

And of course you could add any other code to the main loop, in the same way. There is no need to be “aware” of the interrupt-driven process; it runs quite independently.

That’s the theory, anyway. You might think that, as we did in [lesson 2](#), we could write the code which responds to the pushbutton as:

```
sB_LED = !BUTTON;           // turn on indicator only if button pressed
```

That’s shorter, seems more natural, and is, in theory, equivalent – but in practice it doesn’t work properly in this example. This happens because the code generated by XC8² to implement this statement does not work

² true for version 1.00, running in “Free mode”

correctly if it is interrupted by a routine which also modifies the `sGPIO` union – which of course our ISR, which toggles a bit within `sGPIO` to flash an LED, does. This is despite `sGPIO` being declared as `'volatile'`.

So – in practice, your ISR may interfere in non-obvious ways with your other code, if they are both updating the same variables or structures – especially when you are using a C compiler, where it may not be apparent that the generated code has this susceptibility.

But in general this isn't an issue that you would normally need to worry about. Just remember that it can happen!

The only other change that has to be made to the code in example 1 is to configure both GP1 and GP2 as outputs:

```
TRISIO = 0b111001;           // configure GP1 and GP2 (only) as outputs
```

No changes are needed within the interrupt service routine.

Complete program

Although the changes to the code in example 1 are minor, here is how they fit together:

```

/*****
*   Description:      Lesson 3, example 2
*
*   Demonstrates use of Timer0 interrupt to perform a background task
*   while performing other actions in response to changing inputs
*
*   One LED simply flashes at 1 Hz (50% duty cycle).
*   The other LED is only lit when the pushbutton is pressed.
*
*****/
*
*   Pin assignments:
*   GP1 = "button pressed" indicator LED
*   GP2 = flashing LED
*   GP3 = pushbutton switch (active low)
*
*****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// int reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sB_LED   sGPIO.GP1           // "button pressed" indicator LED (shadow)
#define sF_LED   sGPIO.GP2           // flashing LED (shadow)
#define BUTTON   GPIObits.GP3       // pushbutton

/***** GLOBAL VARIABLES *****/
volatile union {                    // shadow copy of GPIO
    uint8_t      port;
    struct {
        unsigned GP0      : 1;
    };
};

```

```

        unsigned    GP1      : 1;
        unsigned    GP2      : 1;
        unsigned    GP3      : 1;
        unsigned    GP4      : 1;
        unsigned    GP5      : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    GPIO = 0;                // start with all LEDs off
    sGPIO.port = 0;          // update shadow
    TRISIO = 0b111001;      // configure GP1 and GP2 (only) as outputs

    // configure Timer0
    OPTION_REGbits.T0CS = 0; // select timer mode
    OPTION_REGbits.PSA = 1;  // no prescaler (assigned to WDT)
                            // -> increment every 1 us

    // enable interrupts
    INTCONbits.T0IE = 1;    // enable Timer0 interrupt
    ei();                   // enable global interrupts

    /*** Main loop
    for (;;)
    {
        // check and respond to button press
        sB_LED = 0;         // assume button up -> indicator LED off
        if (BUTTON == 0)   // if button pressed (low)
            sB_LED = 1;    // turn on indicator LED

        // continually copy shadow GPIO to port
        GPIO = sGPIO.port;

    } // repeat forever
}

/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt_isr(void)
{
    static uint16_t cnt_t0 = 0; // counts timer0 overflows

    /*** Service Timer0 interrupt
    //
    // TMR0 overflows every 250 clocks = 250 us
    //
    // Flashes LED at 1 Hz by toggling on every 2000th interrupt
    // (every 500 ms)
    //
    // (only Timer0 interrupts are enabled)
    //
    TMR0 += 256-250+3;        // add value to Timer0
                            // for overflow after 250 counts
    INTCONbits.T0IF = 0;    // clear interrupt flag

```

```

// toggle LED every 500 ms
++cnt_t0; // increment interrupt count (every 250 us)
if (cnt_t0 == 500000/250) // on count overflow (every 500 ms),
{
    cnt_t0 = 0; // reset count
    sF_LED = ~sF_LED; // toggle LED (via shadow register)
}
}

```

Example 3: Switch debouncing

[Lesson 1](#) demonstrated one widely-used method of addressing the problem of switch bounce, which was expressed in pseudo-code as:

```

count = 0
while count < max_samples
    delay sample_time
    if input = required_state
        count = count + 1
    else
        count = 0
end

```

The change in switch state is only accepted when the new state has been continually seen for at least some minimum period, for example 20 ms. This debounce period is measured by incrementing a count while sampling the state of the switch, at a steady rate, such as every 1 ms.

We saw in [mid-range lesson 6](#) that this counting algorithm can be readily implemented in an interrupt service routine, which regularly samples the switch and increments a counter whenever the current (or *raw*) state of the switch is different from the last accepted (or *debounced*) state.

That is, if the switch is in a different state from what it used to be, maybe it has “really” changed, or maybe this is just a glitch, or perhaps it’s bouncing, so let’s check a few more times to be sure. When it’s been stable in the new state for some time, we accept this new state as being “real”, and consider the switch to have been debounced.

Although you could have the ISR respond to and act upon switch changes, this isn’t normally done unless the event has to be responded to very quickly; it is generally best to keep the interrupt handling code short, so that the ISR finishes quickly, in case another, perhaps more important, interrupt is pending.

Instead, the ISR would normally use a flag to signal to the main program that an event (such as a change in switch state) has occurred. The main program then polls this flag and responds to the event when it is ready to do so.

In this case, we would need a ‘switch state has changed’ flag.

We also need a flag, or variable, to hold the “debounced”, or most recently accepted state of the switch input. The ISR can then periodically compare the current “raw” switch input with the saved “debounced” input, to determine whether the switch state has changed.

This approach has the advantage that switch changes are detected quickly, while the main program does not have to respond to them immediately.

XC8 implementation

In the assembler example in [mid-range lesson 6](#), the variables holding the debounced pushbutton state and the pushbutton changed flag were defined as:

```
PB_dbstate   res 1           ; bit 3 = debounced pushbutton state
                ; (0 = pressed, 1 = released)
PB_change    res 1           ; bit 3 = flag indicating pushbutton state change
                ; (1 = new debounced state)
```

This definition allocates a whole byte for each variable, even though only a single bit is needed in each case. Bit 3 was used to simplify the assembler code.

However, XC8 provides a ‘bit’ data type, so we may as well make use of it, to simplify the C code, and to allow the compiler to pack these variables into a single byte of data memory (or not, as it sees fit – an advantage of C being that we don’t have to be concerned with these implementation details³).

Since these variables will be updated in the ISR and accessed in the main program, they must be defined as volatile global variables, along with the shadow copy of GPIO:

```
/***** GLOBAL VARIABLES *****/
volatile union {                // shadow copy of GPIO
    uint8_t      port;
    struct {
        unsigned GP0      : 1;
        unsigned GP1      : 1;
        unsigned GP2      : 1;
        unsigned GP3      : 1;
        unsigned GP4      : 1;
        unsigned GP5      : 1;
    };
} sGPIO;

volatile bit     PB_dbstate;    // debounced pushbutton state (1 = released)
volatile bit     PB_change;    // pushbutton state change flag (1 = changed)
```

There is, however, one limitation with the way that bit variables are implemented in XC8 – they cannot be initialised as part of their definition.

That is, we **cannot** write:

```
volatile bit     PB_dbstate = 1; // debounced pushbutton state (1 = released)
volatile bit     PB_change = 0; // pushbutton state change flag (1 = changed)
```

Instead, they must be initialised separately, as part of the initialisation code, before interrupts are enabled (so that they have the correct values when the ISR first runs):

```
// initialise variables
PB_dbstate = 1;           // initial pushbutton state = released
PB_change = 0;           // clear pushbutton change flag (no change)
```

Since the debounce counter is only used within the ISR, it should be defined as being private to (within) the interrupt function, along with the timer interrupt counter:

```
static uint8_t  cnt_t0 = 0;    // counts timer0 interrupts
static uint8_t  cnt_db = 0;    // debounce counter
```

³ This is also a disadvantage of C – by not being aware of how the C compiler builds various constructs, we may not realize that we’re doing things in an inefficient way.

Once again, these variables must be defined as being 'static', so that their values will be preserved between interrupts.

It is a good idea to define the debounce period as a constant, to make it easier to adapt the code for switches with different characteristics:

```
#define MAX_DB_CNT 20/2 // maximum debounce count =
                        // debounce period / sample rate
                        // (20 ms debounce period / 2 ms per sample)
```

(of course it would be cleaner still to define the debounce period and sample rate as constants, and to derive the maximum debounce count and sample timing from them – but in a short program like this it's not difficult to see how these things relate to each other, especially if it is documented in comments, as above).

The debounce routine must be run at some regular interval by the ISR.

In the example in [mid-range lesson 6](#), an interval of 2 ms was used, so we'll do the same here, by incrementing and then testing a counter whenever the Timer0 interrupt is serviced:

```
// sample switch every 2 ms
++cnt_t0; // increment interrupt count (every 250 us)
if (cnt_t0 == 2000/250) // until 2 ms has elapsed
{
    // debounce code goes here
}
```

Within the debounce routine, we must first determine whether the raw pushbutton state has changed since it was last debounced. Since we are using bit variables, this can be written very simply:

```
// compare raw pushbutton with current debounced state
if (BUTTON == PB_dbstate) // if raw state matches last debounced state,
{
    // pushbutton has not changed state
}
else
{
    // pushbutton has changed state
}
```

Where previously the symbol 'BUTTON' had been defined as:

```
// Pin assignments
#define sB_LED sGPIO.GP1 // indicator LED (shadow)
#define BUTTON GPIObits.GP3 // pushbutton
```

Having determined whether the pushbutton's raw state has changed, we need to deal with both possibilities, as allowed for in the `if / else` structure above.

If the pushbutton is still in the last debounced state, all we need to do is reset the debounce counter:

```
cnt_db = 0; // reset debounce count
```


Otherwise, the pushbutton's state has changed. We need to see whether the change is stable, by counting the number of successive times we've seen it in this new state, and then check whether the maximum count has been reached, to determine whether the switch really has changed state (and has finished bouncing):

```
++cnt_db;           // increment debounce count
if (cnt_db == MAX_DB_CNT) // when max count is reached
{
    // accept new state as changed
}
```

If we're accepting that the pushbutton really has changed state, we need to update the variables and flags to reflect this:

```
PB_dbstate = !PB_dbstate; // toggle debounced state
cnt_db = 0;           // reset debounce count
PB_change = 1;        // set pushbutton changed flag
```

The main program can then poll this `PB_change` flag, to see whether the button has changed state:

```
if (PB_change == 1)
{
    // pushbutton has changed state
}
```

But since this variable is a binary flag, the code can be more clearly written as:

```
if (PB_change)
{
    // pushbutton has changed state
}
```

If the button has changed state, we then need to refer to the `PB_dbstate` variable, to see whether it the new state is "up" or "down" (pressed); we only want to toggle the LED when the button is pressed, not when it is released, so we could write:

```
if (PB_change)
{
    // pushbutton has changed state, so check for button press
    if (PB_dbstate == 0)
    {
        // pushbutton has been pressed (low)
    }
}
```

Or, if you prefer, you can write this much more succinctly as:

```
if (PB_change && !PB_dbstate)
{
    // button state has changed and is pressed (low)
}
```

As ever, it's a question of personal style.

Once we've determined that the button has been pressed, we can toggle the LED, using the shadow copy of GPIO, as we've done before:

```
sB_LED = ~sB_LED;           // toggle LED (via shadow register)
```

And finally, now that we've detected and responded to the button press, we need to clear the state change flag, to be ready for the next change:

```
PB_change = 0;           // clear button change flag
```

And that's all.

It's relatively complex, compared with the equivalent code in the example in [lesson 2](#), but most of that complexity is "hidden" in the ISR; the code in the main program loop is quite simple, making it easier to do more within the main program, without having to poll and debounce switches – something that the ISR can take care of in the background.

Complete program

Here is the complete "toggle an LED on pushbutton press" program:

```

/*****
 *
 * Description: Lesson 3, example 3
 *
 * Demonstrates use of Timer0 interrupt to implement
 * counting debounce algorithm
 *
 * Toggles LED when the pushbutton is pressed (high -> low)
 *
 *****/
 *
 * Pin assignments:
 * GP1 = indicator LED
 * GP3 = pushbutton (active low)
 *
 *****/
#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// int reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sB_LED    sGPIO.GP1           // indicator LED (shadow)
#define BUTTON   GPIObits.GP3       // pushbutton

/***** CONSTANTS *****/
#define MAX_DB_CNT  20/2 // max debounce count = debounce period / sample rate
                       // (20 ms debounce period / 2 ms per sample)

/***** GLOBAL VARIABLES *****/
volatile union { // shadow copy of GPIO
    uint8_t      port;
    struct {
        unsigned GP0    : 1;
        unsigned GP1    : 1;
        unsigned GP2    : 1;
        unsigned GP3    : 1;
    };
}

```

```

        unsigned    GP4      : 1;
        unsigned    GP5      : 1;
    };
} sGPIO;

volatile bit    PB_dbstate;    // debounced pushbutton state (1 = released)
volatile bit    PB_change;    // pushbutton state change flag (1 = changed)

/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    GPIO = 0;                // start with all LEDs off
    sGPIO.port = 0;         // update shadow
    TRISIO = ~(1<<1);      // configure GP1 (only) as an output

    // configure Timer0
    OPTION_REGbits.T0CS = 0; // select timer mode
    OPTION_REGbits.PSA = 1;  // no prescaler (assigned to WDT)
                            // -> increment every 1 us

    // initialise variables
    PB_dbstate = 1;         // initial pushbutton state = released
    PB_change = 0;         // clear pushbutton change flag (no change)

    // enable interrupts
    INTCONbits.T0IE = 1;    // enable Timer0 interrupt
    ei();                   // enable global interrupts

    //*** Main loop
    for (;;)
    {
        // check for debounced button press
        if (PB_change && !PB_dbstate) // if PB state changed and pressed (low)
        {
            sB_LED = ~sB_LED;        // toggle LED (via shadow register)
            PB_change = 0;           // clear button change flag
        }

        // continually copy shadow GPIO to port
        GPIO = sGPIO.port;

    } // repeat forever
}

/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt_isr(void)
{
    static uint8_t cnt_t0 = 0;    // counts timer0 interrupts
    static uint8_t cnt_db = 0;    // debounce counter

    //*** Service Timer0 interrupt
    //
    // TMR0 overflows every 250 clocks = 250 us
    //
    // Debounces pushbutton:
    // samples every 2 ms (every 8th interrupt)

```

```

//  -> PB_dbstate = debounced state
//      PB_change  = change flag (1 = new debounced state)
//
//  (only Timer0 interrupts are enabled)
//
TMR0 += 256-250+3;           // add value to Timer0
                               //   for overflow after 250 counts
INTCONbits.T0IF = 0;       // clear interrupt flag

// Debounce pushbutton
//  use counting algorithm: accept change in state
//  only if new state is seen a number of times in succession

// sample switch every 2 ms
++cnt_t0;                    // increment interrupt count (every 250 us)
if (cnt_t0 == 2000/250)     // until 2 ms has elapsed
{
    cnt_t0 = 0;              // reset interrupt count

    // compare raw pushbutton with current debounced state
    if (BUTTON == PB_dbstate) // if raw PB matches debounced state,
        cnt_db = 0;          // reset debounce count
    else // else raw pushbutton has changed state
    {
        ++cnt_db;            // increment debounce count
        if (cnt_db == MAX_DB_CNT) // when max count is reached
        {
            // accept new state as changed:
            PB_dbstate = !PB_dbstate; // toggle debounced state
            cnt_db = 0;           // reset debounce count
            PB_change = 1;       // set pushbutton changed flag
                                   // (polled and cleared in main)
        }
    }
}
}
}

```

Example 4: Switch debouncing while flashing an LED

Since the previous example on switch debouncing was built on the framework of the earlier LED flashing examples, it's not difficult to add the LED flashing code back into the interrupt service routine, showing how a single timer-driven interrupt can be used to schedule multiple concurrent tasks.

In the assembler example in [mid-range lesson 6](#), a variable was used in the Timer0 interrupt service routine to count periods of 2 ms each (the debounce sample period), to generate a 500 ms time base, used to toggle the LED. This method (building on the existing 2 ms time base) was used in order to simplify the code, with only one additional 8-bit variable being needed.

XC8 implementation

Although we could take the same approach – adding a single 8-bit variable to count 2 ms periods – the ease of handling 16-bit quantities in C means that there is little reason to do so. If you were really hard pressed to fit your variables into the available data memory, you might consider ways to save a byte here and there, although in that case, you're probably better off either using a bigger PIC or programming in assembler. We'll continue to take approaches which seem comfortable and natural from a C perspective, even if they are not necessarily the most efficient – because the emphasis when programming in C is a little different from programming in assembler.

So, as we did for the 1 Hz flashing example above, we'll define a static 16-bit variable, within the interrupt function, for the counter used to generate the 500 ms time base:

```
static uint8_t      db_t_cnt = 0;    // debounce sample timebase counter
static uint8_t      db_s_cnt = 0;    // debounce sample counter
static uint16_t     fl_t_cnt = 0;    // LED flash timebase counter
```

Note that the counter variables from the previous example have been renamed, for clarity and consistency; we now have two counters, generating two independent time bases within the same timer interrupt service routine, so it needs to be clear which is which.

And then, either before or after the debounce routine in the ISR, we need to add some code to increment the counter to generate the 500 ms time base, and flash the LED:

```
++fl_t_cnt;                // increment interrupt count (every 250 us)
if (fl_t_cnt == 500000/250) // until 500 ms has elapsed
{
    fl_t_cnt = 0;          // reset interrupt count
    sF_LED = ~sF_LED;     // toggle LED (via shadow register)
}
```

Complete interrupt service routine

Most of the code is the same as the previous example, except for the counter variable definition and initialisation, shown above. The main loop is unchanged. But here is the new interrupt service routine, so that you can see how the LED toggling code fits in after the debounce routine:

```
/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    static uint8_t      db_t_cnt = 0;    // debounce sample timebase counter
    static uint8_t      db_s_cnt = 0;    // debounce sample counter
    static uint16_t     fl_t_cnt = 0;    // LED flash timebase counter

    /*** Service Timer0 interrupt
    //
    // TMR0 overflows every 250 clocks = 250 us
    //
    // Debounces pushbutton:
    //   samples every 2 ms (every 8th interrupt)
    //   -> PB_dbstate = debounced state
    //   PB_change = change flag (1 = new debounced state)
    //
    // Flashes LED at 1 Hz by toggling on every 2000th interrupt
    //   (every 500 ms)
    //
    // (only Timer0 interrupts are enabled)
    //
    TMR0 += 256-250+3;          // add value to Timer0
                                //   for overflow after 250 counts
    INTCONbits.T0IF = 0;      // clear interrupt flag

    // Debounce pushbutton
    //   use counting algorithm: accept change in state
    //   only if new state is seen a number of times in succession
    //
    // sample switch every 2 ms
    ++db_t_cnt;                // increment interrupt count (every 250 us)
    if (db_t_cnt == 2000/250) // until 2 ms has elapsed
    {
        db_t_cnt = 0;          // reset interrupt count
```

```

// compare raw pushbutton with current debounced state
if (BUTTON == PB_dbstate) // if raw PB matches current debounce state,
    db_s_cnt = 0;          // reset debounce count
else                       // else raw pushbutton has changed state
{
    ++db_s_cnt;           // increment debounce count
    if (db_s_cnt == MAX_DB_CNT) // when max count is reached
    {
        PB_dbstate = !PB_dbstate; // toggle debounced state
        db_s_cnt = 0;           // reset debounce count
        PB_change = 1;         // set pushbutton changed flag
                                // (polled and cleared in main)
    }
}

// Flash LED (toggle every 500 ms)
//
++fl_t_cnt; // increment interrupt count (every 250 us)
if (fl_t_cnt == 500000/250) // until 500 ms has elapsed
{
    fl_t_cnt = 0; // reset interrupt count
    sF_LED = ~sF_LED; // toggle LED (via shadow register)
}
}

```

Comparisons

Here is the resource usage summary for the “flash LED while toggling on pushbutton press” programs:

Flash+Toggle_LED

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	98	77	8
XC8 (Free mode)	55	154	13

The C source code continues to be around half as long as the assembly source, while the (unoptimised) code generated by XC8 (running in “Free mode”) is more than twice as large as the assembly version.

External Interrupts

Although polling input pins for changes is effective in many cases, especially in user interfaces, where the human user won’t notice a delay of a few milliseconds before a button press is responded to, some situations require a more immediate response.

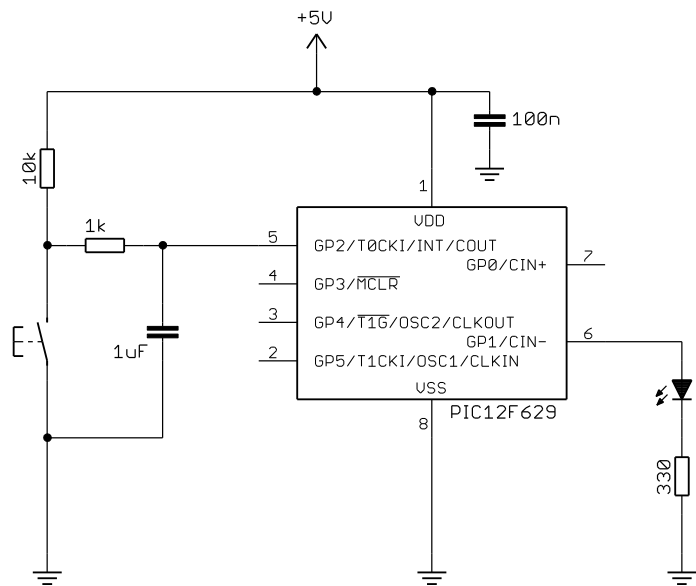
For a very fast response to a digital signal, the external interrupt, INT (which shares its pin with GP2) can be used. This pin is *edge-triggered*, meaning that an interrupt will be triggered (if enabled) by a rising or falling transition of the input signal.

Example 5: Using a pushbutton to trigger an external interrupt

To show how to use external interrupts, we can toggle an LED whenever the external interrupt is triggered by a pushbutton press, using the circuit from [mid-range lesson 6](#), shown (with the reset switch and its pull-up resistor omitted for clarity) on the right.

As explained in that lesson, the capacitor connected across the switch is used, in conjunction with the two resistors, to debounce the pushbutton, because it is difficult to implement software debouncing for an edge-triggered interrupt, while retaining a fast response.

To implement this circuit with the [Gooligum training board](#), close jumpers JP3, JP7 and JP12 to enable the 10 k Ω pull-up resistors on \overline{MCLR} and GP2 and the LED on GP1. You also need to add a 1 μ F capacitor (supplied with the board) between GP2 and ground. You can do this via pins 13 ('GP/RA/RB2') and 16 ('GND') on the 16-pin expansion header. There should be no need to use the solderless breadboard – simply plug the capacitor directly into these header pins.



This simple RC filter approach can be used because the 12F629's INT input is a Schmitt trigger type, as explained in [baseline assembler lesson 4](#).

The assembler code in [mid-range lesson 6](#) configured the external interrupt, so that it would be triggered by a falling edge (high \rightarrow low transition) on the INT pin (caused by the pushbutton being pressed), by clearing the INTEDG bit in the OPTION register:

```
; configure external interrupt
banksel OPTION_REG
bcf    OPTION_REG,INTEDG    ; trigger on falling edge
```

We then enabled the external interrupt, by setting the INTE bit in the INTCON register:

```
; enable interrupts
movlw  1<<GIE|1<<INTE    ; enable external and global interrupts
movwf  INTCON
```

(also setting GIE, as always, to globally enable interrupts)

Within the ISR, the only actions which needed to be taken were to clear the INTF interrupt flag (to indicate that the external interrupt has been serviced) and to toggle the LED on GP1:

```
bcf    INTCON,INTF        ; clear interrupt flag

; toggle LED
movlw  1<<nB_LED          ; toggle indicator LED
xorwf  sGPIO,f           ; using shadow register
```

The shadow register was copied to GPIO in the main loop, as in the earlier examples.

XC8 implementation

Implementing these steps using XC8 is quite straightforward, being very similar to what we have done before.

Firstly, to select the type of transition to trigger the external interrupt:

```
// configure external interrupt
OPTION_REGbits.INTEDG = 0;      // trigger on falling edge
```

Then to enable the external interrupt:

```
// enable interrupts
INTCONbits.INTE = 1;           // enable external interrupt
ei();                           // enable global interrupts
```

And finally to service the external interrupt:

```
INTCONbits.INTF = 0;           // clear interrupt flag

// toggle LED
sB_LED = ~sB_LED;              // (via shadow register)
```

Complete program

Here is how these code fragments (along with code from the previous examples) fit together:

```

/*****
 *
 * Description: Lesson 3, example 5
 *
 * Demonstrates use of external interrupt (INT pin)
 *
 * Toggles LED when pushbutton on INT is pressed
 * (high -> low transition)
 *
 *****/
 *
 * Pin assignments:
 * GP1 = indicator LED
 * INT = pushbutton (active low)
 *
 *****/
#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sB_LED  sGPIO.GP1           // indicator LED (shadow)

/***** GLOBAL VARIABLES *****/
volatile union {                  // shadow copy of GPIO
    uint8_t      port;
    struct {
        unsigned GP0      : 1;

```



```

        unsigned    GP1      : 1;
        unsigned    GP2      : 1;
        unsigned    GP3      : 1;
        unsigned    GP4      : 1;
        unsigned    GP5      : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    GPIO = 0;                // start with all LEDs off
    sGPIO.port = 0;         // update shadow
    TRISIO = ~(1<<1);      // configure GP1 (only) as an output

    // configure external interrupt
    OPTION_REGbits.INTEDG = 0; // trigger on falling edge

    // enable interrupts
    INTCONbits.INTE = 1;     // enable external interrupt
    ei();                    // enable global interrupts

    /*** Main loop
    for (;;)
    {
        // continually copy shadow GPIO to port
        GPIO = sGPIO.port;

    } // repeat forever
}

/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt_isr(void)
{
    /*** Service external interrupt
    //
    // Triggered on high -> low transition on INT pin
    // caused by externally debounced pushbutton press
    //
    // Toggles LED on every high -> low transition
    //
    // (only external interrupts are enabled)
    //
    INTCONbits.INTF = 0;     // clear interrupt flag

    // toggle LED
    sB_LED = ~sB_LED;       // (via shadow register)
}

```

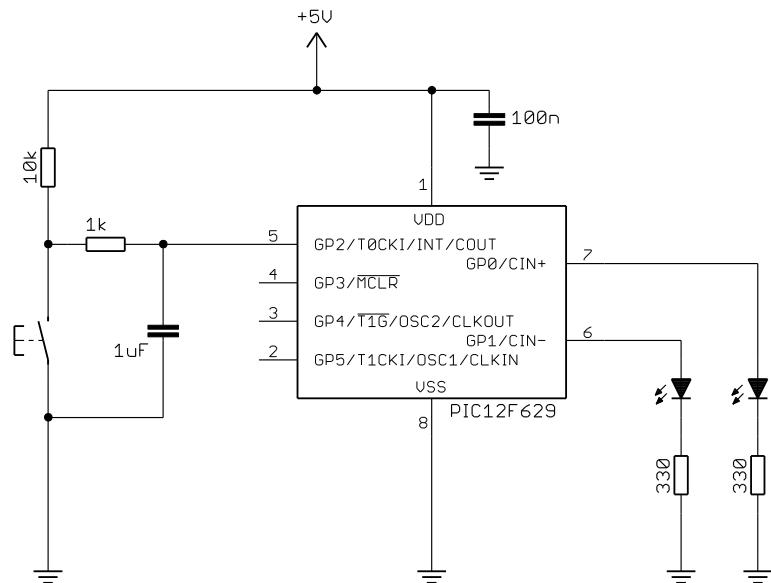
Example 6: Multiple interrupt sources

So far we've only used a single interrupt source, but it is common for more than one source to be active; for example, one or more timers scheduling background tasks, while servicing events such as external interrupts.

To demonstrate this, we can combine the two interrupt sources used in this lesson, with a Timer0 interrupt flashing one LED, while the external interrupt is used to toggle another LED.

This means adding an LED to the circuit in the previous example, as shown on the right.

If you have the [Gooligum training board](#), leave it set up as in the last example, but also close jumper JP11 to enable the LED on GP0.



We'll flash the LED on GP0 at 1 Hz, and toggle the LED on GP1 whenever the pushbutton is pressed, as we did in [mid-range lesson 6](#).

The program in the example in [mid-range lesson 6](#) was put together by re-using routines from the previous LED flashing and external interrupt examples.

Of course, both interrupt sources had to be enabled:

```
; enable interrupts
movlw  1<<GIE|1<<T0IE|1<<INTE  ; enable external, Timer0
movwf  INTCON                    ; and global interrupts
```

And code had to be added to the interrupt service routine, checking the interrupt flags to determine which source had triggered the interrupt, and then branching to the appropriate service handler:

```
; *** Identify interrupt source
btfsc  INTCON,INTF              ; external
goto   ext_int
btfsc  INTCON,T0IF              ; Timer0
goto   t0_int
goto   isr_end                  ; none of the above, so exit
```

In this way, only one interrupt source will be serviced, each time an interrupt is triggered. If more than one interrupt is pending (more than one interrupt flag is set), another interrupt will be triggered, immediately after the ISR exits, and the next interrupt source will be serviced the next time the ISR is run.

Since only one source was to be serviced when an interrupt was triggered, a ‘goto’ instruction was added to the end of each service handler, to skip to the end of the ISR:

For example:

```
ext_int ; *** Service external interrupt
;
;   Triggered on high -> low transition on INT pin
;   caused by externally debounced pushbutton press
;
;   Toggles LED on every high -> low transition
;
bcf     INTCON,INTF           ; clear interrupt flag

; toggle LED
movlw  1<<nB_LED             ; toggle indicator LED
xorwf  sGPIO,f               ; using shadow register
goto   isr_end
```

XC8 implementation

When checking for multiple interrupt sources, using C, it seems most natural to use a series of ‘if’ statements, each testing an interrupt flag, and executing the corresponding service handler if that interrupt flag is set.

For example:

```
// Service all triggered interrupt sources

if (INTCONbits.INTF)
{
    // External interrupt handler goes here
}

if (INTCONbits.T0IF)
{
    // Timer0 interrupt handler goes here
}
```

With this structure, every pending interrupt source will be serviced when an interrupt is triggered. This is different from the assembly version given above, where only one source is serviced per interrupt.

The C version is perhaps clearer and has slightly less overhead (since fewer interrupts may be triggered overall), but in practice the difference is negligible.

In both approaches, the highest priority interrupt source should be serviced first – in this case we consider an external interrupt to more important (should be serviced more quickly) than a timer overflow, but that’s something only you can decide, in the context of your application.

The actual interrupt handlers are the same as before, so they are easy to “plug in” to this framework.

The only other addition needed is to enable all the interrupt sources:

```
// enable interrupts
INTCONbits.T0IE = 1;           // enable Timer0 interrupt
INTCONbits.INTE = 1;          // enable external interrupt
ei();                          // enable global interrupts
```

Complete program

Here is the complete “toggle LED via external interrupt while flashing LED via timer interrupt” program, so that you can see how these pieces fit together:

```

/*****
*
* Description: Lesson 3, example 6
*
* Demonstrates handling of multiple interrupt sources
*
* Toggles an LED when pushbutton on INT is pressed
* (high -> low transition triggering external interrupt)
* while another LED flashes at 1 Hz (driven by Timer0 interrupt)
*
*****/
*
* Pin assignments:
* GP0 = flashing LED
* GP1 = indicator LED
* INT = pushbutton (active low)
*
*****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sF_LED  sGPIO.GP0           // flashing LED (shadow)
#define sB_LED  sGPIO.GP1           // "button pressed" indicator LED (shadow)

/***** GLOBAL VARIABLES *****/
volatile union {                    // shadow copy of GPIO
    uint8_t      port;
    struct {
        unsigned GP0      : 1;
        unsigned GP1      : 1;
        unsigned GP2      : 1;
        unsigned GP3      : 1;
        unsigned GP4      : 1;
        unsigned GP5      : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    GPIO = 0;                // start with all LEDs off
    sGPIO.port = 0;         // update shadow
    TRISIO = 0b111100;     // configure GP0 and GP1 (only) as outputs

```

```

// configure Timer0
OPTION_REGbits.T0CS = 0;           // select timer mode
OPTION_REGbits.PSA = 1;           // no prescaler (assigned to WDT)
                                   // -> increment every 1 us

// configure external interrupt
OPTION_REGbits.INTEDG = 0;        // trigger on falling edge

// enable interrupts
INTCONbits.T0IE = 1;              // enable Timer0 interrupt
INTCONbits.INTE = 1;              // enable external interrupt
ei();                              // enable global interrupts

/***/ Main loop
for (;;)
{
    // continually copy shadow GPIO to port
    GPIO = sGPIO.port;

} // repeat forever
}

/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt_isr(void)
{
    static uint16_t    fl_t_cnt = 0; // LED flash timebase counter

    // Service all triggered interrupt sources

    if (INTCONbits.INTF)
    {
        /***/ Service external interrupt
        //
        // Triggered on high -> low transition on INT pin
        // caused by externally debounced pushbutton press
        //
        // Toggles LED on every high -> low transition
        //
        INTCONbits.INTF = 0;          // clear interrupt flag

        // toggle LED
        sB_LED = ~sB_LED;            // (via shadow register)
    }

    if (INTCONbits.T0IF)
    {
        /***/ Service Timer0 interrupt
        //
        // TMR0 overflows every 250 clocks = 250 us
        //
        // Flashes LED at 1 Hz by toggling on every 2000th interrupt
        // (every 500 ms)
        //
        TMR0 += 256-250+3;            // add value to Timer0
                                   // for overflow after 250 counts
        INTCONbits.T0IF = 0;         // clear interrupt flag

        // Flash LED (toggle every 500 ms)
        //

```

```
    ++fl_t_cnt;                // incr interrupt count (every 250 us)
    if (fl_t_cnt == 500000/250) // until 500 ms has elapsed
    {
        fl_t_cnt = 0;          // reset interrupt count
        sF_LED = ~sF_LED;     // toggle LED (via shadow register)
    }
}
```

Summary

These examples have demonstrated that XC8 can be used to implement interrupts, in a very straightforward way. Because the compiler takes care of many of the details, such as saving and restoring processor context, transparently, the C source code can be quite simple and succinct.

On the other hand, we also saw that, because the compiler hides implementation details, it may be harder to uncover and avoid situations where the interrupt code interferes in a non-obvious way with operations performed on variables or structures which are updated in both the ISR and your main code.

In other words, there can be a price to pay for apparent simplicity...

Nevertheless, interrupts are too useful for tasks such as background processes (such as flashing an LED), while responding to and processing events (such as detecting and debouncing key presses), to ignore – they need to remain an important part of our toolkit, whether we're using C or not.

We'll see more examples as topics are introduced in future lessons.

The next interrupt source we'll look at is "interrupt on change", which is commonly used to wake the PIC from sleep mode. It is covered in the [next lesson](#), along with the watchdog timer.

Introduction to PIC Programming

Programming Mid-Range PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 4: Interrupt-on-change, Sleep Mode and the Watchdog Timer

This lesson revisits material from [mid-range lesson 7](#), looking at the mid-range PIC architecture's power-saving sleep mode, its ability to generate interrupts and/or wake from sleep when an input changes state and the watchdog timer – generally used to automatically restart a crashed program, but also useful for periodically waking the PIC from sleep, for low-power operation.

One again, the examples are re-implemented using Microchip's XC8 compiler¹ (running in "Free mode"), introduced in [lesson 1](#).

In summary, this lesson covers:

- Interrupt-on-change
- Sleep mode (power down)
- Wake-up on change (power up on input change)
- The watchdog timer
- Periodic wake from sleep

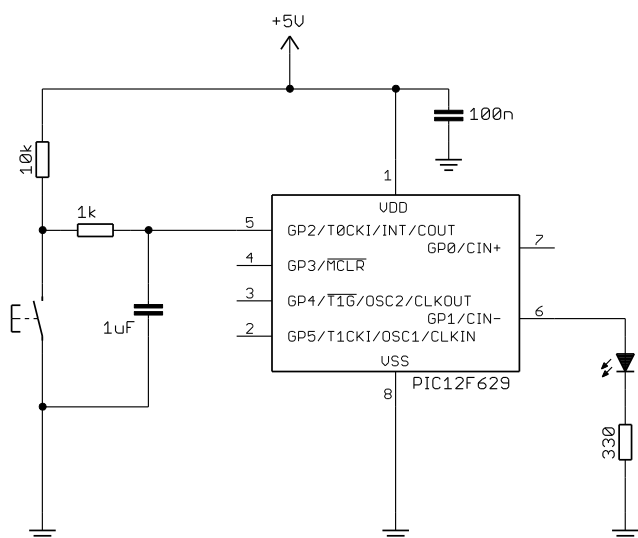
Interrupt-on-change

As we saw in [mid-range lesson 7](#), mid-range PICs provide a port change interrupt facility, which, on the 12F629, is available on every GPIO pin.

This feature is similar to the external interrupt facility covered in [lesson 3](#), except that a port change interrupt will be triggered by any change (not just one type of transition) on any of the pins for which it is enabled. This makes it more flexible (being available on more pins), but also more difficult to deal with correctly, as we shall see in the examples in this section.

The first example uses the circuit on the right to demonstrate how to use interrupt-on-change to respond to a single, externally debounced input.

If you are using the [Gooligum training board](#), close jumpers JP3, JP7 and JP12 to enable the 10 kΩ pull-up resistors on $\overline{\text{MCLR}}$ (not shown here)



¹ Available as a free download from www.microchip.com.

and GP2 and the LED on GP1. You can add the 1 μ F capacitor (supplied with the board) between GP2 and ground via pins 13 ('GP/RA/RB2') and 16 ('GND') on the 16-pin expansion header. There should be no need to use the solderless breadboard – simply plug the capacitor directly into these header pins.

GP2 is used here because, on the 12F629, it has a Schmitt-trigger input, allowing the simple RC filter to provide effective hardware debouncing, as explained in [baseline assembler lesson 4](#).

To enable a pin for interrupt-on-change, the corresponding bit must be set in the IOC register. This was done in [mid-range lesson 7](#) by:

```
banksel IOC           ; enable interrupt-on-change
bsf      IOC,nBUTTON ; on pushbutton input
```

(where 'nBUTTON' is a constant which has been set to '2')

Before actually enabling port change interrupts, it is necessary to either read or write to the port to clear any existing *mismatch condition*, to prevent any false triggering.

The port change interrupt can then be enabled by setting the GPIE bit in the INTCON register.

This was done in assembler by:

```
; enable interrupts
movlw   1<<GIE|1<<GPIE ; enable port change and global interrupts
movwf   INTCON
```

(note that global interrupts are also being enabled here, by setting the GIE bit)

In the interrupt handler, we must clear the port mismatch condition which triggered this interrupt.

In [mid-range lesson 7](#) this was done by reading the port. And (as for all interrupts), we must also clear the corresponding interrupt flag, GPIF:

```
banksel GPIO
movf    GPIO,w           ; clear mismatch condition
bcf     INTCON,GPIF     ; clear interrupt flag
```

Since we want to toggle the LED on GP1 each time the pushbutton is pressed, but not when it is released, we need to check whether the switch is up or down (this is different from the situation with external interrupts, which are only triggered on one type of transition).

This was implemented in assembler as:

```
; toggle LED only on button press
btfsc   GPIO,nBUTTON    ; is button down?
goto    isr_end
movlw   1<<nB_LED        ; if so, toggle indicator LED
xorwf   sGPIO,f         ; using shadow register
```

(where 'nB_LED' is a constant which has been set to '1')

The shadow register was copied to GPIO in the main loop, as in the earlier examples.

XC8 implementation

Implementing these steps using XC8 is quite straightforward, using techniques we have seen before.

Enabling interrupt-on-change on GP2 is simply:

```
IOCbits.IOC2 = 1;           // enable IOC on GP2 input
```


To enable the port change and global interrupts, we have:

```
// enable interrupts
INTCONbits.GPIE = 1;           // enable port change interrupt
ei();                          // enable global interrupts
```

In the interrupt handler, it is best to explicitly clear the mismatch condition (by reading GPIO) at the start of the routine, instead of relying on this occurring as a side-effect of statements in the body of the handler, which may be changed later.

This can be done by:

```
GPIO;                          // read GPIO to clear mismatch condition
```

“GPIO” is an expression which evaluates to the value of the contents of GPIO, but does nothing with it.

In general, the compiler’s optimiser will discard any such “do nothing” statements.

However, GPIO is declared as a ‘volatile’ variable in the processor header file. We’ve seen that this qualifier warns the compiler that the value of this variable may change at any time, to prevent the optimiser from eliminating apparently redundant references to it. It also ensures that, when the variable’s name is used on its own in this way, the compiler will generate code which reads the variable’s memory location and discards the result, which is exactly what we want.

We must also clear the port interrupt flag, to indicate that this interrupt has been serviced:

```
INTCONbits.GPIF = 0;          // clear interrupt flag
```

Finally, we need to check the status of the pushbutton, and toggle the LED only if the button is pressed (meaning that GP2 is low):

```
// toggle LED only on button press
if (!BUTTON)                    // if button is down
    sB_LED = ~sB_LED;          // toggle LED (via shadow register)
```

Complete program

Here is how these code fragments fit into a working program:

```
/******
 *
 * Description:    Lesson 4, example 1
 *
 * Demonstrates use of interrupt-on-change interrupts
 * (without software debouncing)
 *
 * Toggles LED when pushbutton is pressed (high -> low transition)
 *
 *****/
 *
 * Pin assignments:
 * GP1 = indicator LED
 * GP2 = pushbutton (externally debounced, active low)
 *
 *****/

#include <xc.h>
#include <stdint.h>
```

```

/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define SB_LED   sGPIO.GP1           // indicator LED (shadow)
#define BUTTON  GPIObits.GP2       // pushbutton

/***** GLOBAL VARIABLES *****/
volatile union {                    // shadow copy of GPIO
    uint8_t      port;
    struct {
        unsigned GP0    : 1;
        unsigned GP1    : 1;
        unsigned GP2    : 1;
        unsigned GP3    : 1;
        unsigned GP4    : 1;
        unsigned GP5    : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    GPIO = 0;                // start with all LEDs off
    sGPIO.port = 0;         // update shadow
    TRISIO = ~(1<<1);      // configure GP1 (only) as an output
    IOCbts.IOC2 = 1;       // enable IOC on GP2 input

    // enable interrupts
    INTCONbits.GPIE = 1;    // enable port change interrupt
    ei();                  // enable global interrupts

    /*** Main loop
    for (;;)
    {
        // continually copy shadow GPIO to port
        GPIO = sGPIO.port;

    } // repeat forever
}

/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt_isr(void)
{
    /*** Service port change interrupt
    //
    // Triggered on any transition on IOC-enabled input pin
    // caused by externally debounced pushbutton press
    //
    // Toggles LED on every high -> low transition
    //

```

```

// (only port change interrupts are enabled)
//
GPIO; // read GPIO to clear mismatch condition
INTCONbits.GPIF = 0; // clear interrupt flag

// toggle LED only on button press
if (!BUTTON) // if button is down
    sB_LED = ~sB_LED; // toggle LED (via shadow register)
}

```

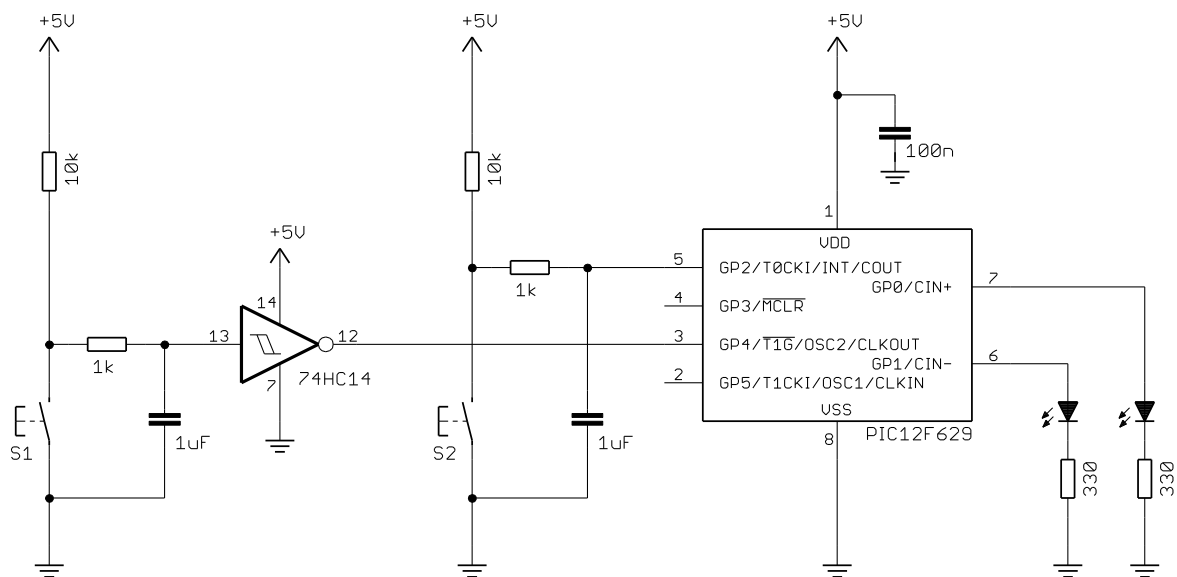
Example 2: Interrupt-on-change (multiple inputs)

This example demonstrates how to handle the situation where interrupt-on-change is enabled on more than one input pin.

The basic difficulty with handling this situation is that there are no flags to indicate which input has changed; the GPIF flag can tell you that at least one pin enabled for IOC has changed, but not which pin it was.

So when a port change interrupt occurs, we need to deduce which pin(s) have changed, by reading GPIO and comparing the current state to the last recorded state. That means that the ISR (where the port change interrupt is handled) needs to keep track of the state of GPIO, and update that “last state” record, every time a change is detected, to be ready for the next time.

We’ll use the circuit from the corresponding example in [mid-range lesson 7](#), shown (with the reset switch and pull-up omitted for clarity) below:



If you have the [Gooligum training board](#), you can use the additional components supplied with your board to build this circuit on the solderless breadboard, connecting them to signals on the 16-pin header: GP4 input on pin 3 (‘GP/RA/RB4’), GP2 input on pin 13 (‘GP/RA/RB2’) and ground and +5 V on pins 15 (‘+V’) and 16 (‘GND’) – see the illustration in [mid-range lesson 7](#). You should also close JP3, JP7, JP11 and JP12 to enable the pull-up resistors on $\overline{\text{MCLR}}$ (not shown here) and GP2 and the LEDs on GP0 and GP1.

Each pushbutton toggles an LED: S1 controls the LED on GP1, and S2 controls the LED on GP0.

To simplify the software, both buttons are externally debounced, and since the only Schmitt-trigger GP input on the 12F629 is GP2, an external Schmitt-trigger inverter is used to drive GP4.

Thus, the operation of S1 is inverted with respect to S2; the software will have to take this difference into account.

XC8 implementation

As in the last example, to make the code more maintainable, it is good practice to define symbols represent the pins being used:

```
// Pin assignments
#define SB1_LED sGPIO.GP0 // "button 1 pressed" indicator LED (shadow)
#define SB2_LED sGPIO.GP1 // "button 2 pressed" indicator LED (shadow)
#define nPB1 2 // pushbutton 1 (ext debounce, active low) on GP2
#define nPB2 4 // pushbutton 2 (ext debounce, active high) on GP4
```

Note that the pushbutton pins have been defined as numeric constants, representing pin numbers, because it simplifies the change detection code (see below).

Given that we must keep track of the “last state” of GPIO, to compare with the current state when a port change is detected, and that this state will need to be initialised in the main code, but accessed and updated in the ISR, we need to declare it as a volatile global variable (along with the shadow copy of GPIO, which is also accessed in both the ISR and the main code):

```
/****** GLOBAL VARIABLES *****/
volatile union { // shadow copy of GPIO
    uint8_t port;
    struct {
        unsigned GP0 : 1;
        unsigned GP1 : 1;
        unsigned GP2 : 1;
        unsigned GP3 : 1;
        unsigned GP4 : 1;
        unsigned GP5 : 1;
    };
} sGPIO;

volatile uint8_t lGPIO; // last state of GPIO (for change detection)
```

In the initialisation code, we need to initialise and configure the port, before updating the “last state” variable, so that everything is in sync:

```
// configure port
GPIO = 0; // start with all LEDs off
sGPIO.port = 0; // update shadow
TRISIO = 0b111100; // configure GP0 and GP1 (only) as outputs
lGPIO = GPIO; // update last port state (for pin change detection)
```

Why bother reading GPIO, when we just cleared it?

Why not just write:

```
GPIO = 0; // start with all LEDs off
sGPIO = 0; // update shadow
lGPIO = 0; // and last state (NOTE: THIS WILL NOT WORK!)
TRISIO = 0b111100; // configure GP0 and GP1 (only) as outputs
```

This approach will not, in general, work, because the value read from an input pin depends on the external signal applied to the pin; if an input pin is being held high externally, clearing the port register (GPIO) won't have any effect – it will still read as a '1'.

So the only way to be sure of the current state of **GPIO** is to read it.

Having done so, we can enable interrupt-on-change for both inputs, with:

```
IOC = 1<<nPB1|1<<nPB2; // enable interrupt-on-change on pushbuttons 1 and 2
```

A useful side-effect of reading **GPIO** is that it clears any existing IOC mismatch condition, so we can now safely go ahead and enable the port change interrupt:

```
// enable interrupts
INTCONbits.GPIE = 1; // enable port change interrupt
ei(); // enable global interrupts
```

As usual, the main loop does nothing more than continually update **GPIO** from the shadow register:

```
/***/ Main loop
for (;;)
{
    // continually copy shadow GPIO to port
    GPIO = sGPIO.port;

} // repeat forever
```

Meanwhile, the ISR updates the shadow copy of **GPIO**, whenever a port change occurs (triggering an interrupt).

Within the ISR, it is best to take a “snapshot” of the current state of **GPIO**, and use this to determine which pins have changed, instead of referring back continually to **GPIO** itself, in case an input changes while the interrupt handler is running (leading to inconsistent results).

So we declare a variable within the ISR function, so hold this current state:

```
void interrupt isr(void)
{
    uint8_t    cGPIO; // current state of GPIO (used by IOC handler)

    // IOC handler goes here...
}
```

When servicing the port change interrupt, we begin by clearing the interrupt flag, as usual:

```
INTCONbits.GPIF = 0; // clear interrupt flag
```

There is no need to explicitly clear the IOC mismatch here, because **GPIO** is read in the very next statement:

```
cGPIO = GPIO; // save current GPIO state
```

Next we need to determine which pin(s) have changed, by comparing the current state of **GPIO** with the last recorded state.

This can be done by XORing the current and last states. Since an XOR results in a ‘1’ only where the inputs differ, the result will be all ‘0’s, except for those bits corresponding to any pins which have changed.

We could write this as:

```
changes = 1GPIO ^ cGPIO // XOR current with last state to detect changes
```

but there is actually no need to introduce another variable; the only time we need to reference the last state (1GPIO) is here, to deduce which pins have changed, and, having done so, there is no need to refer back to 1GPIO again, until it is updated at the end of the ISR.

So, to save data memory, it is possible to write the XOR result back to 1GPIO with:

```
1GPIO ^= cGPIO;           // XOR with last state to detect changes
```

The 1GPIO variable will now contain '1's only in bit positions where the current state differs from the last state, corresponding to pins that have changed.

We can then use this to check whether each pushbutton input has changed, for example:

```
if (1GPIO & 1<<nPB1)      // if button 1 changed
{
    // handle button 1 input change...
}
```

But since we only want to toggle the LED when the pushbutton has pressed, we must check not only that the pushbutton input has changed, but that the button is down, which we can do with nested if statements:

```
// toggle LED 1 only on button 1 press (active low)
if (1GPIO & 1<<nPB1)      // if button 1 changed
    if (!(cGPIO & 1<<nPB1)) // and if button 1 is down (low)
    {
        sB1_LED = ~sB1_LED; // toggle LED 1 (via shadow register)
    }
```

Alternatively, this can be written as a single if statement, using a logical AND expression:

```
// toggle LED 1 only on button 1 press (active low)
if ((1GPIO & 1<<nPB1)      // if button 1 changed
    && !(cGPIO & 1<<nPB1)) // and button 1 is down (low)
    {
        sB1_LED = ~sB1_LED; // toggle LED 1 (via shadow register)
    }
```

Either form is acceptable; both generate the same (efficient) code, so which you use only a question of personal programming style.

Looking at these logical expressions, you may conclude that it would also be possible to replace the logical AND ('&&') with a bitwise AND ('&') and condense the expression to:

```
if (1GPIO & cGPIO & 1<<nPB1) // if button 1 changed and down
{
    sGPIO ^= 1<<nB1_LED;      // toggle LED 1 using shadow register
}
```

However, although this works, in terms of program logic (the expressions are, after all, logically the same), it is less clear and generates less efficient code. This is a case where writing more obscure code is counter-productive – it's simply a bad idea.

We can then write a very similar construct for the second pushbutton, but with the logic for testing "button down" inverted because this signal is active high, not low:

```
// toggle LED 2 only on button 2 press (active high)
if ((1GPIO & 1<<nPB2)      // if button 2 changed
    && (cGPIO & 1<<nPB2)) // and button 2 is down (high)
    {
        sB2_LED = ~sB2_LED; // toggle LED 2 (via shadow register)
    }
```

Finally, before exiting the interrupt handler, we must save the current state of GPIO as the new “last state”, so that the next input change can be properly detected:

```
// update last GPIO state (for next time)
lGPIO = cGPIO;                // new "last state" = current
```

Complete program

Here is how these code fragments fit together, to form the complete “interrupt-on-change with multiple inputs” example program:

```

/*****
*   Description:      Lesson 4, example 2
*
*   Demonstrates handling of multiple interrupt-on-change interrupts
*   (without software debouncing)
*
*   Toggles LED on GP0 when pushbutton on GP2 is pressed
*   (high -> low transition)
*   and LED on GP1 when pushbutton on GP4 is pressed
*   (low -> high transition)
*
*****/
*   Pin assignments:
*   GP0 = indicator LED 1
*   GP1 = indicator LED 2
*   GP2 = pushbutton 1 (externally debounced, active low)
*   GP4 = pushbutton 2 (externally debounced, active high)
*
*****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define SB1_LED sGPIO.GP0    // "button 1 pressed" indicator LED (shadow)
#define SB2_LED sGPIO.GP1    // "button 2 pressed" indicator LED (shadow)
#define nPB1      2          // pushbutton 1 (ext debounce, active low) on GP2
#define nPB2      4          // pushbutton 2 (ext debounce, active high) on GP4

/***** GLOBAL VARIABLES *****/
volatile union {                // shadow copy of GPIO
    uint8_t      port;
    struct {
        unsigned GP0      : 1;
        unsigned GP1      : 1;
        unsigned GP2      : 1;
        unsigned GP3      : 1;
        unsigned GP4      : 1;
        unsigned GP5      : 1;
    };
} sGPIO;

```

```

volatile uint8_t    lGPIO;           // last state of GPIO (for change detection)

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    GPIO = 0;                // start with all LEDs off
    sGPIO.port = 0;         // update shadow
    TRISIO = 0b111100;     // configure GP0 and GP1 (only) as outputs
    lGPIO = GPIO;          // update last port state (for pin change detection)
    IOC = 1<<nPB1|1<<nPB2; // enable interrupt-on-change on pushbuttons 1 and 2

    // enable interrupts
    INTCONbits.GPIE = 1;    // enable port change interrupt
    ei();                   // enable global interrupts

    /*** Main loop
    for (;;)
    {
        // continually copy shadow GPIO to port
        GPIO = sGPIO.port;

    } // repeat forever
}

/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    uint8_t    cGPIO;        // current state of GPIO (used by IOC handler)

    /*** Service port change interrupt
    //
    // Triggered on any transition on IOC-enabled input pin
    // caused by externally debounced pushbutton press
    //
    // Toggles LED1 on every high -> low transition of PB1
    // and LED2 on every low -> high transition of PB2
    //
    // (only port change interrupts are enabled)
    //
    INTCONbits.GPIF = 0;    // clear interrupt flag

    // determine which pins have changed
    cGPIO = GPIO;          // save current GPIO state
                           // (GPIO read clears mismatch condition)
    lGPIO ^= cGPIO;        // XOR with last state to detect changes

    // toggle LED 1 only on button 1 press (active low)
    if ((lGPIO & 1<<nPB1) // if button 1 changed
        && (!(cGPIO & 1<<nPB1))) // and button 1 is down (low)
    {
        sB1_LED = ~sB1_LED; // toggle LED 1 (via shadow register)
    }

    // toggle LED 2 only on button 2 press (active high)
    if ((lGPIO & 1<<nPB2) // if button 2 changed

```



```

    && (cGPIO & 1<<nPB2))        // and button 2 is down (high)
    {
        sB2_LED = ~sB2_LED;      // toggle LED 2 (via shadow register)
    }

    // update last GPIO state (for next time)
    lGPIO = cGPIO;                // new "last state" = current
}

```

Sleep Mode

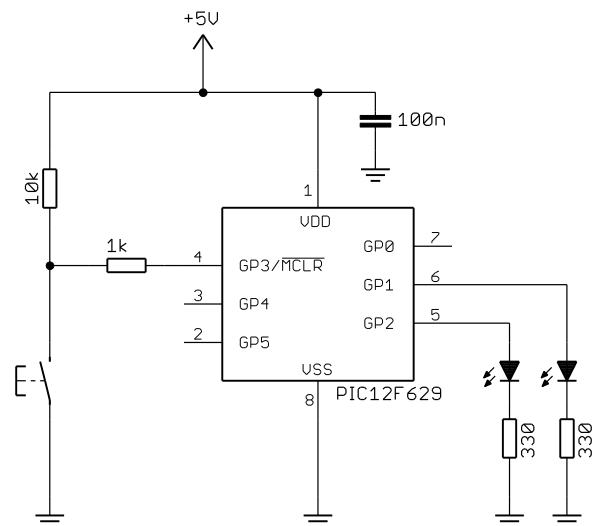
As explained in [mid-range lesson 7](#), the mid-range PICs can be placed into a power-saving standby, or sleep mode, using the assembler instruction 'sleep'.

In this mode, the PIC12F629 will typically draw only a few nanoamps (or less), when all of the power-consuming facilities have been disabled and the output pins are not supplying any current.

This was demonstrated using the circuit on the right.

To implement it using the [Gooligum training board](#), close jumpers JP3, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP1 and GP2.

To demonstrate to yourself that power consumption really is reduced when the PIC enters sleep mode, you would have to use an external power supply, instead of using your PICkit 2 or PICkit 3 to power the circuit. You can then place a multimeter in-line with the power supply, to measure the supply current.



The LED on GP1 is initially turned on, and then when the pushbutton is pressed, the LED is turned off (reducing power consumption) before placing the PIC permanently into sleep mode (effectively shutting it down).

The following assembler code was used:

```

    ; turn on LED
    banksel GPIO
    bsf     LED

    ; wait for button press
wait_lo btfsc  BUTTON          ; wait until button low
        goto  wait_lo

    ; go into standby mode
    sleep          ; enter sleep mode

    goto  $        ; (this instruction should never run)

```

(where 'BUTTON' and 'LED' are symbols representing GP3 and GP1 respectively)

XC8 implementation

To place the PIC into sleep mode, XC8 provides a 'SLEEP()' macro.

It is defined in the "pic.h" header file (called from the "xc.h" file we've included at the start of each program), as:

```
#define SLEEP() asm("sleep")
```

'asm()' is a XC8 statement which embeds a single assembler instruction, in-line, in the C source code. But since 'SLEEP()' is provided as a standard macro, it makes sense to use it, instead of the 'asm()' statement.

Complete program

The following program shows how the XC8 'SLEEP()' macro is used:

```

/*****
*   Description:   Lesson 4, example 3
*
*   Demonstrates sleep mode
*
*   Turn on LED, wait for button pressed, turn off LED, then sleep
*
*****/
*
*   Pin assignments:
*       GP1 = indicator LED
*       GP3 = pushbutton (active low)
*
*****/

#include <xc.h>

/***** CONFIGURATION *****/
// int reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define LED      GPIObits.GP1    // indicator LED on GP1
#define nLED     1               // (port bit 1)
#define BUTTON   GPIObits.GP3    // pushbutton on GP3 (active low)

/***** MAIN PROGRAM *****/
void main()
{
    //***** Initialisation

    // configure port
    TRISIO = ~(1<<nLED);        // configure LED pin (only) as output

    //***** Main code

    // turn on LED
    LED = 1;

    // wait for button press
    while (BUTTON == 1)        // wait until button low
        ;
}

```

```

// go into standby (low power) mode
LED = 0;           // turn off LED
SLEEP();          // enter sleep mode

for (;;)          // (this loop should never execute)
    ;
}

```

Wake-up from sleep

As discussed in [mid-range lesson 7](#), mid-range PICs can be woken from sleep mode in a number of ways:

- Any device reset, such as an external reset signal on the $\overline{\text{MCLR}}$ pin (if enabled)
- Watchdog timer timeout (see the section on the watchdog timer, later in this lesson)
- Any enabled interrupt source which can set its interrupt flag while in sleep mode

Since the PIC's oscillator (clock) does not run in sleep mode, interrupt sources which require the clock to function, such as Timer0, cannot be used wake the device from sleep. However, external (INT pin) and port change interrupts (and others that we will see in later lessons) can be used to wake up a mid-range PIC.

The following example looks at how to use the port change interrupt to wake a PIC from sleep mode; the method for using an external interrupt is essentially the same, but is of course limited to the INT pin.

Example 4: Using interrupt-on-change for wake-up from sleep

In [baseline assembler lesson 7](#), we saw that the baseline architecture includes a “wake-up on change” feature. Its mid-range equivalent is the interrupt-on-change facility, introduced above.

“Interrupt-on-change” can be used to wake the device from sleep, even if interrupts are not enabled. If port change interrupts are enabled ($\text{GPIE} = 1$), but global interrupts are disabled ($\text{GIE} = 0$), then the device will wake from sleep when an IOC-enabled input changes, but no interrupt will occur. Program execution simply continues with the instruction following the `sleep` instruction, or, if using XC8, the statement following the `'SLEEP()'` macro.

If port change interrupts are enabled ($\text{GPIE} = 1$) and global interrupts are enabled ($\text{GIE} = 1$), if an IOC-enabled input changes while the PIC is in sleep mode, the device will wake from sleep, execute the instruction following `sleep`, and then enter the interrupt service routine.

If you want the PIC to execute the ISR immediately after it wakes from sleep, you need to enable interrupts and place a `nop` (“do nothing” – available in XC8 as a `'NOP()'` macro) instruction immediately following the `sleep` instruction.

If you are using other interrupts (such as Timer0) in your program, but don't want to have to deal with executing the ISR as the device wakes from sleep, simply disable interrupts (clear GIE – which can be done in XC8 with the `'di()'` macro) before entering sleep mode.

In any case, if $\text{GPIE} = 1$, the PIC will wake if the value of any IOC-enabled input changes while it is in sleep mode.

It is important to clear the GPIF flag before entering sleep mode, or else the PIC will wake immediately.

Note: You should read the input pins configured for interrupt-on-change just prior to entering sleep mode, and clear GPIF . Otherwise, if the value at an IOC-enabled pin had changed since the last time it was read, the PIC will wake immediately upon entering sleep mode, as the input value would be seen to be different from that last read.

It is also important to ensure that any input which will be used to trigger a wake-up is stable before entering sleep mode.

This means that any switch used as a “soft” on/off switch must be debounced both as soon as the PIC has been restarted (in case the switch is still bouncing) and prior to entering sleep mode (in case a bounce causes the PIC to wake).

In this example, we want to wake-up the PIC and turn on an LED when the button is pressed, and then turn off the LED and place the PIC into sleep mode when the button is pressed again.

The necessary sequence is:

```
do
  turn on LED
  wait for stable button high
  wait for button low
  turn off LED
  wait for stable button high
  clear GPIF
  sleep
forever    // repeat from the beginning
```

XC8 implementation

The following code, which uses the debounce macro defined in [lesson 2](#), implements the sequence of steps given above:

```
/** Initialisation */

// configure port
TRISIO = ~(1<<nLED);    // configure LED pin (only) as output

// configure Timer0 (for DbnceHi() macro)
OPTION_REGbits.T0CS = 0;    // select timer mode
OPTION_REGbits.PSA = 0;    // assign prescaler to Timer0
OPTION_REGbits.PS = 0b111;  // prescale = 256
                          // -> increment every 256 us

// configure interrupt-on-change
IOC |= 1<<nBUTTON;    // enable IOC on pushbutton input
INTCONbits.GPIE = 1;    // enable wake-up (interrupt) on port change

/** Main loop */
for (;;)
{
  // turn on LED
  LED = 1;

  // wait for stable button high
  // (in case it is still bouncing after wakeup)
  DbnceHi(BUTTON);

  // wait for button press
  while (BUTTON == 1)    // wait until button low
    ;

  // go into standby (low power) mode
  LED = 0;    // turn off LED
  DbnceHi(BUTTON);    // wait for stable button release
```

```

        INTCONbits.GPIF = 0;           // clear port change interrupt flag
        SLEEP();                       // enter sleep mode
    }
}

```

(the labels 'LED', 'nLED', 'BUTTON' and 'nBUTTON' are defined earlier in the program, as usual)

This code does essentially the same thing as the “toggle an LED” programs developed in lessons [1](#) and [2](#), except that in this case, when the LED is off, the PIC is drawing negligible power.

Watchdog Timer

As described in [mid-range lesson 7](#), the watchdog timer is free-running counter which, if enabled, operates independently of the program running on the PIC. It is typically used to avoid program crashes, where your application enters a state it will never return from, such as a loop waiting for a condition that will never occur. If the watchdog timer overflows, the PIC is reset, restarting your program – hopefully allowing it to recover and operate normally.

To avoid this “WDT reset” from occurring, your program must periodically reset, or clear, the watchdog timer before it overflows.

This watchdog time-out period on the mid-range PICs is nominally 18 ms, but can be extended to a maximum of 2.3 seconds by assigning the prescaler to the watchdog timer (in which case the prescaler is no longer available for use with Timer0).

The watchdog timer can also be used to regularly wake the PIC from sleep mode, perhaps to sample and log an environmental input (say a temperature sensor), for low power operation.

Example 5a: Enabling the watchdog timer and detecting WDT resets

To illustrate how the watchdog timer allows the PIC to recover from a crash, we'll use a simple program which turns on an LED for 1.0 s, turns it off again, and then enters an endless loop (simulating a crash).

If the watchdog timer is disabled, the loop will never exit and the LED will remain off. But if the watchdog timer is enabled, with a period of 2.3 s, the program should restart itself after 2.3s, and the LED will flash: on for 1.0 s and off for 1.3 s (approximately).

We saw in [mid-range lesson 7](#) that the watchdog timer is controlled by the WDTE bit in the processor configuration word: setting WDTE to '1' enables the watchdog timer.

Since the configuration word cannot be accessed by programs running on the PIC (it can only be written to when the PIC is being programmed), **the watchdog timer cannot be enabled or disabled at runtime**. It can only be configured to be 'on' or 'off' when the PIC is programmed.

The assembler examples in that lesson included the following construct, to make it easy to select whether the watchdog timer is enabled or disabled when the code is built:

```

#define WATCHDOG ; define to enable watchdog timer

IFDEF WATCHDOG
    ; ext reset, no code or data protect, no brownout detect,
    ; watchdog, power-up timer, 4Mhz int clock
    __CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_ON &
    _PWRTE_ON & _INTRC_OSC_NOCLKOUT
ELSE
    ; ext reset, no code or data protect, no brownout detect,
    ; no watchdog, power-up timer, 4Mhz int clock
    __CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
    _PWRTE_ON & _INTRC_OSC_NOCLKOUT
ENDIF

```

To set the watchdog time-out period to the maximum of 2.3 seconds, the prescaler was assigned to the watchdog timer, with a prescale ratio of 1:128 ($18\text{ ms} \times 128 = 2.3\text{ s}$), by:

```
movlw    1<<PSA | b'111' ; assign prescaler to WDT (PSA = 1)
                               ; prescale = 128 (PS = 111)
banksel  OPTION_REG      ; -> WDT timeout = 2.3 s
movwf    OPTION_REG
```

If you want your program to behave differently when restarted by a watchdog time-out, test the $\overline{\text{TO}}$ flag in the STATUS register: it is cleared to '0' only when a WDT reset has occurred.

The example in [mid-range lesson 7](#) used this approach to turn on an “error” LED, to indicate if a restart was due to a WDT reset:

```
***** Initialisation
    ; configure port
    movlw    ~(1<<nF_LED|1<<nW_LED) ; configure LED pins as outputs
    banksel  TRISIO
    movwf    TRISIO
    ; configure watchdog timer prescaler
    movlw    1<<PSA | b'111' ; assign prescaler to WDT (PSA = 1)
                               ; prescale = 128 (PS = 111)
    banksel  OPTION_REG      ; -> WDT timeout = 2.3 s
    movwf    OPTION_REG

***** Main code
    banksel  GPIO
    btfss    STATUS,NOT_TO    ; if WDT timeout has occurred,
    bsf      GPIO,nW_LED      ; turn on "WDT" LED

    bsf      GPIO,nF_LED      ; turn on "flashing" LED

    DelayMS  1000              ; delay 1s

    banksel  GPIO              ; turn off "flashing" LED
    bcf      GPIO,nF_LED

    goto     $                  ; wait forever
```

XC8 implementation

Since the watchdog timer is controlled by a configuration bit, the only change we need to make to enable it is to use a different `__CONFIG()` statement, with the symbol 'WDTE_ON' replacing 'WDTE_OFF'.

A construct very similar to that in the assembler example can be used to select between processor configurations:

```
#ifdef WATCHDOG
    // ext reset, no code or data protect, no brownout detect,
    // watchdog, power-up timer enabled, int RC clock
    __CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
              WDTE_ON & PWRTE_OFF & FOSC_INTRCIO);
#else
    // ext reset, no code or data protect, no brownout detect,
    // no watchdog, power-up timer enabled, int RC clock
    __CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
              WDTE_OFF & PWRTE_OFF & FOSC_INTRCIO);
#endif
```

Assigning the prescaler to the watchdog timer and selecting a prescale ratio of 128:1 is done by:

```
OPTION_REGbits.PSA = 1;           // assign prescaler to WDT
OPTION_REGbits.PS = 0b111;       // prescale = 128
                                // -> WDT timeout = 2.3 s
```

To check for a WDT timeout reset, the \overline{TO} flag can be tested directly, using:

```
if (!STATUSbits.nTO)             // if WDT timeout has occurred,
    W_LED = 1;                   // turn on "error" LED
```

Note that the test condition is inverted, using ‘!’, since this flag is “active” when clear.

Complete program

Here is the complete program, showing how the above code fragments are used:

```
/******
 *
 * Description: Lesson 4, example 5a
 *
 * Demonstrates watchdog timer
 * plus differentiation of WDT time-out from POR reset
 *
 * Turn on LED for 1s, turn off, then enter endless loop
 * If WDT enabled, processor resets after 2.3s
 * Turns on WDT LED to indicate WDT reset
 *
 *****/
 *
 * Pin assignments:
 * GP1 = flashing LED
 * GP2 = WDT-reset indicator LED
 *
 *****/
#include <xc.h>

#define _XTAL_FREQ 4000000 // oscillator frequency for _delay()

/***** CONFIGURATION *****/
#define WATCHDOG // define to enable watchdog timer

#ifdef WATCHDOG
    // ext reset, no code or data protect, no brownout detect,
    // watchdog, power-up timer enabled, int RC clock
    __CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
        WDTE_ON & PWRTE_OFF & FOSC_INTRCIO);
#else
    // ext reset, no code or data protect, no brownout detect,
    // no watchdog, power-up timer enabled, int RC clock
    __CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
        WDTE_OFF & PWRTE_OFF & FOSC_INTRCIO);
#endif

// Pin assignments
#define F_LED GPIObits.GP1 // "flashing" LED on GP1
#define nF_LED 1 // (port bit 1)
#define W_LED GPIObits.GP2 // WDT LED to indicate WDT time-out reset
#define nW_LED 2 // (port bit 2)
```

```

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation ***/

    // configure port
    TRISIO = ~(1<<nF_LED|1<<nW_LED);    // configure LED pins as outputs

    // configure watchdog timer
    OPTION_REGbits.PSA = 1;            // assign prescaler to WDT
    OPTION_REGbits.PS = 0b111;        // prescale = 128
                                        // -> WDT timeout = 2.3 s

    /*** Main code ***/

    // test for WDT-timeout reset
    if (!STATUSbits.nTO)                // if WDT timeout has occurred,
        W_LED = 1;                    // turn on "error" LED

    // flash LED
    F_LED = 1;                          // turn on "flash" LED
    __delay_ms(1000);                    // delay 1 sec
    F_LED = 0;                          // turn off "flash" LED

    // wait forever
    for (;;)
        ;
}

```

Example 5b: Clearing the watchdog timer

Normally, you will want to prevent watchdog timer overflows; a WDT reset should only happen when something has gone wrong.

To avoid WDT resets, the watchdog timer has to be regularly cleared. This is typically done by inserting a 'clrwdt' instruction within the program's "main loop", and within any subroutine which may, in normal operation, not complete within the watchdog timer period.

To demonstrate the effect of clearing the watchdog timer, a 'clrwdt' instruction was added into the endless loop in the example in [mid-range lesson 7](#):

```

;***** Main code
    banksel GPIO                ; turn on LED
    bsf      GPIO,LED

    DelayMS 1000                ; delay 1 sec

    banksel GPIO                ; turn off LED
    bcf      GPIO,LED

loop   clrwdt                    ; clear watchdog timer
       goto  loop                ; repeat forever

```

With the 'clrwdt' instruction in place, the watchdog timer never overflows, so the PIC is never restarted by a WDT reset, and the LED remains turned off (until the power is cycled), whether the watchdog timer is enabled or not.

XC8 implementation

XC8 provides a 'CLRWDT()' macro, defined in the "pic.h" header file as:

```
#define CLRWDT() asm("clrwdt")
```

That is, the 'CLRWDT()' macro simply inserts a 'clrwdt' instruction into the code.

Using this macro, the assembler code above can be implemented with XC8 as follows:

```
LED = 1;                // turn on LED

__delay_ms(1000);      // delay 1 sec

LED = 0;                // turn off LED

for (;;)                // repeatedly clear watchdog timer forever
    CLRWDT();
```

Example 6: Periodic wake from sleep

As explained in [mid-range lesson 7](#), the watchdog timer is can also be used to periodically wake the PIC from sleep mode, typically to read some inputs, take some action and then return to sleep mode, saving power. This can be combined with wake-up on pin change, allowing immediate response to some inputs, such as a button press, while periodically checking others.

To illustrate this, the example in [mid-range lesson 7](#) converted the main code in the first watchdog timer example into a loop, incorporating the 'sleep' instruction:

```
main_loop
    banksel GPIO        ; turn on LED
    bsf    LED

    DelayMS 1000        ; delay 1 sec

    banksel GPIO        ; turn off LED
    bcf    LED

    sleep                ; enter sleep mode (until WDT time-out)

    goto   main_loop    ; repeat forever
```

With the watchdog timer enabled, with a period of 2.3 s, the LED is on for 1 s, and then off for 1.3 s, as in the earlier example. But this time the PIC is in sleep mode while the LED is off, conserving power.

XC8 implementation

In a similar way, we can convert the main code in example 5, above, into a loop – dropping the WDT timeout test, and adding a SLEEP() macro:

```
for (;;)
{
    LED = 1;                // turn on LED

    __delay_ms(1000);      // delay 1 sec

    LED = 0;                // turn off LED

    SLEEP();                // enter sleep mode (until WDT time-out)
}
```

Complete program

Here is how this new main loop fits into the code:

```

/*****
*
* Description: Lesson 4, example 6
*
* Demonstrates periodic wake from sleep, using the watchdog timer
*
* Turn on LED for 1s, turn off, then sleep
* LED stays off if watchdog not enabled,
* flashes (1s on, 2.3s off) if WDT enabled
*
*****/
*
* Pin assignments:
* GP1 = indicator LED
*
*****/

#include <xc.h>

#define _XTAL_FREQ 4000000 // oscillator frequency for _delay_ms()

/***** CONFIGURATION *****/
#define WATCHDOG // define to enable watchdog timer

#ifndef WATCHDOG
// ext reset, no code or data protect, no brownout detect,
// watchdog, power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
WDTE_ON & PWRTE_OFF & FOSC_INTRCIO);
#else
// ext reset, no code or data protect, no brownout detect,
// no watchdog, power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
WDTE_OFF & PWRTE_OFF & FOSC_INTRCIO);
#endif

// Pin assignments
#define LED GPIObits.GP1 // indicator LED on GP1
#define n_LED 1 // (port bit 1)

/***** MAIN PROGRAM *****/
void main()
{
//*** Initialisation ***/

// configure port
TRISIO = ~(1<<n_LED); // configure LED pin (only) as output

// configure watchdog timer
OPTION_REGbits.PSA = 1; // assign prescaler to WDT
OPTION_REGbits.PS = 0b111; // prescale = 128
// -> WDT timeout = 2.3 s

//*** Main loop ***/
for (;;)

```

```
{
    LED = 1;                // turn on LED
    __delay_ms(1000);      // delay 1 sec
    LED = 0;                // turn off LED
    SLEEP();                // enter sleep mode (until WDT time-out)
}
```

Summary

We have seen in this lesson that the interrupt-on-change, sleep mode, wake-up on change, and watchdog timer features of the mid-range PIC architecture can be configured and used effectively in C programs, using the XC8 compiler.

The [next lesson](#) revisits material from [mid-range assembler lesson 8](#), briefly covering some of the hardware-related features of the 12F629 (and most other mid-range PICs), such as brown-out detection and the available oscillator (clock) options.

Introduction to PIC Programming

Programming Mid-Range PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 5: Reset, Power and Clock Options

[Mid-range assembler lesson 8](#) looked at some of the more “hardware-related” aspects of the mid-range PIC architecture, including clock sources, the power-on reset conditions needed to successfully power-up a mid-range PIC, and brown-out resets and detection. This lesson covers the same topics, re-implementing the examples using Microchip’s XC8 compiler¹ (running in “Free mode”), as usual.

However, there is no to repeat all of the theory here, so you may wish to refer back to [mid-range lesson 8](#) for more detail.

In summary, this lesson covers:

- Oscillator (clock) options
- Power-on reset (POR)
- Power-up timer (PWRT)
- Brown-out detection (BOD)

Oscillator (Clock) Options

Although it is often appropriate to use the internal RC oscillator as the processor clock source, there are some situations where it is more appropriate to use some external clock circuitry, for reasons such as:

- *Greater accuracy and stability.*
A crystal or ceramic resonator is significantly more accurate than the internal RC oscillator, with less frequency drift due to temperature and voltage variations.
- *Generating a specific frequency.*
For example, as we saw in [lesson 2](#), the signal from a 32.768 kHz crystal can be readily divided down to 1 Hz. Or, to produce accurate timing for RS-232 serial data transfers, a crystal frequency such as 1.843200 MHz can be used, since it is an exact multiple of common bit rates, such as 38400 or 9600 ($1843200 = 48 \times 38400 = 192 \times 9600$).
- *Synchronising with other components.*
Clocking a number of devices from a common source, so that their outputs change synchronously, may simplify your design – although you need to be careful; clock signals which are subject to varying delays in different parts of your circuit will not be properly synchronised (a phenomenon known as *clock skew*), leading to unpredictable results.

¹ Available as a free download from www.microchip.com.

Another approach is to make the PIC's clock available externally, so that other components can be synchronised with it.

- *Lower power consumption.*

At a given supply voltage, PICs draw less current when they are clocked at a lower speed. Power consumption can be minimised by running the PIC at the slowest practical clock speed and power supply voltage. And for many applications, a high clock rate is unnecessary.

- *Faster operation.*

Most mid-range PICs can operate at a clock rate of up to 20 MHz, while the internal RC oscillator generally runs at only 4 or 8 MHz. If you need more speed than the internal oscillator can provide, you need to use a crystal or other external clock source.

Mid-range PICs support a number of clock, or oscillator, configurations, allowing, through appropriate oscillator selection, any of these goals to be met (but not necessarily all at once – low power consumption and high frequencies don't mix!)

The following table summarises the oscillator configuration options available for the PIC12F629, and the corresponding MPASM and XC8 symbols:

FOSC<2:0>	MPASM symbol	XC8 symbol	Oscillator configuration
000	_LP_OSC	FOSC_LP	LP oscillator
001	_XT_OSC	FOSC_XT	XT oscillator
010	_HS_OSC	FOSC_HS	HS oscillator
011	_EC_OSC	FOSC_EC	EC oscillator
100	_INTRC_OSC_NOCLKOUT	FOSC_INTRCIO	Internal RC oscillator + GP4
101	_INTRC_OSC_CLKOUT	FOSC_INTRCCLK	Internal RC oscillator + CLKOUT
110	_EXTRC_OSC_NOCLKOUT	FOSC_EXTRCIO	External RC oscillator + GP4
111	_EXTRC_OSC_CLKOUT	FOSC_EXTRCCLK	External RC oscillator + CLKOUT

Internal RC oscillator

Until now we've been using the 'FOSC_INTRCIO' configuration, where the internal RC oscillator provides a (nominally) 4 MHz processor clock (FOSC), driving the execution of instructions at approximately 1 MHz, and every pin is available for I/O.

In the 'FOSC_INTRCCLK' configuration, the instruction clock (FOSC/4) is output on the CLKOUT pin, to allow external devices to be synchronised with the PIC's internal RC clock.

Since, on the 12F629, CLKOUT shares pin 3, GP4 cannot be used for I/O in 'FOSC_INTRCCLK' mode.

You can use an oscilloscope to look at the signal on CLKOUT in 'FOSC_INTRCCLK' mode, but to verify that this signal is indeed the instruction clock, it's useful to toggle another pin as quickly as possible, for comparison with CLKOUT, using a simple program such as:

```

/*****
*   Description:      Lesson 5, example 1
*
*   Demonstrates CLKOUT function in Internal RC oscillator mode
*
*   Toggles a pin as quickly as possible
*   for comparison with 1 MHz CLKOUT signal
*****/

```

```

*****
*
*   Pin assignments:
*   GP2      = fast-changing output
*   CLKOUT   = 1 MHz clock output
*
*****/

#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, 4 Mhz int clock with CLKOUT
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCCLK);

// Pin assignments
#define OUT      GP2          // fast-changing output

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation ***/

    // configure port
    TRISIO = 0;              // configure all pins (except GP3 and GP4/CLKOUT)
                           // as outputs

    /*** Main loop ***/
    for (;;)
    {
        OUT = ~OUT;         // toggle output pin as fast as possible
    }
}

```

The internal RC oscillator with CLKOUT configuration was selected by:

```

// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, 4 Mhz int clock with CLKOUT
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCCLK);

```

To toggle the GP2 pin as quickly as possible, the main loop was made as tight as possible:

```

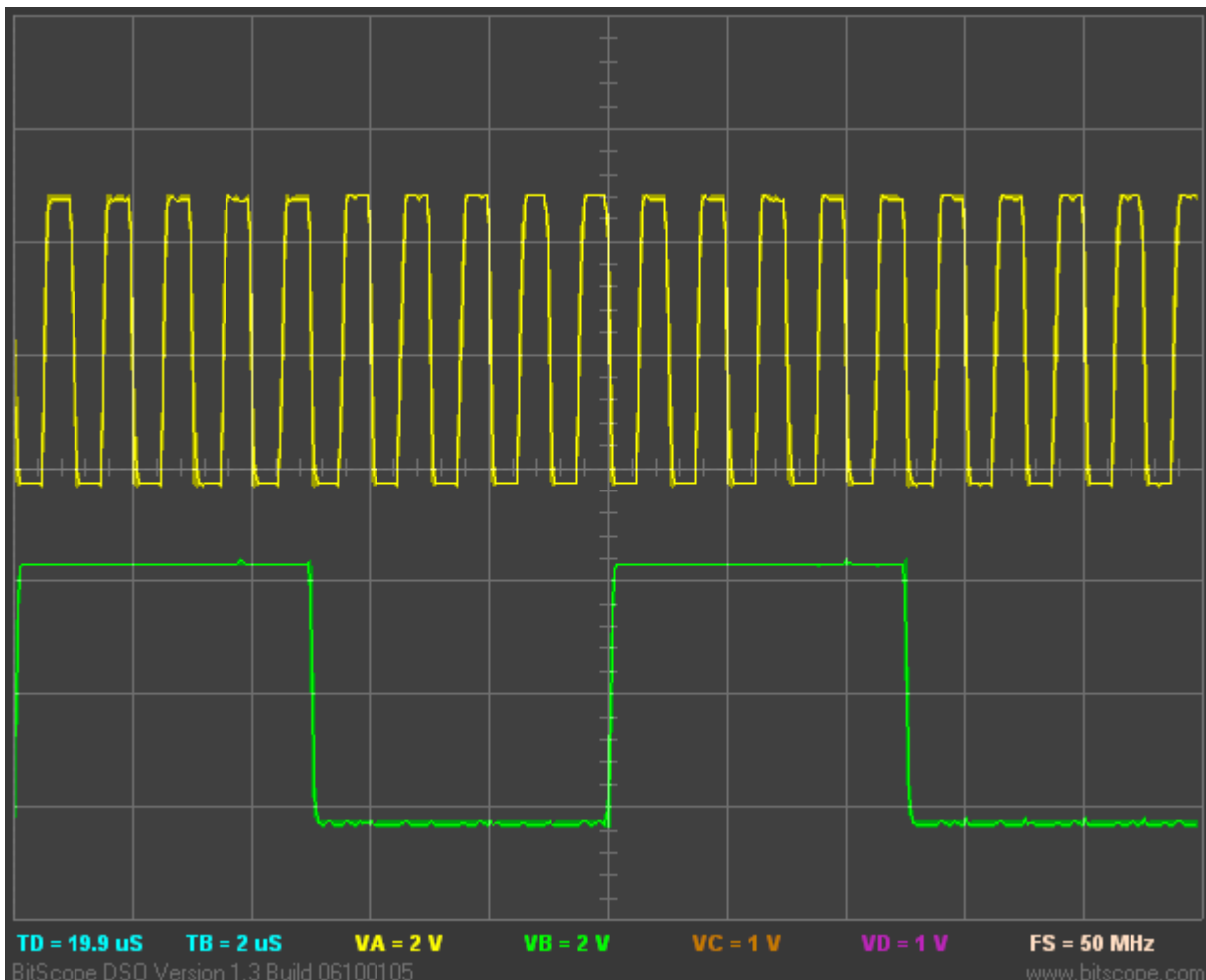
for (;;)
{
    OUT = !OUT;          // toggle output pin as fast as possible
}

```

The XC8 compiler, running in “Free mode”, generates code which toggles GP2 every five cycles, i.e. every 5 μ s, giving an output frequency of 100 kHz.

This is not as fast as we were able to toggle the pin in the example in [mid-range lesson 8](#) – demonstrating that for best results in time-critical code, it may be necessary to use assembler.

This is apparent in the following oscilloscope plot:



The top trace is the instruction clock signal on CLKOUT, which, as you can see, has a period very close to 1 μ s, giving a frequency of 1 MHz, as expected.

The bottom trace is the signal on GP2, which changes state every five instruction cycles, also as expected. Note that the transitions on GP2 are aligned with the falling edge of the instruction clock on CLKOUT.

These signals are available on pins 3 ('GP/RA/RB4') and 13 ('GP/RA/RB2') of the 16-pin header on the [Gooligum training board](#); the ground reference is pin 16 ('GND').

External clock input

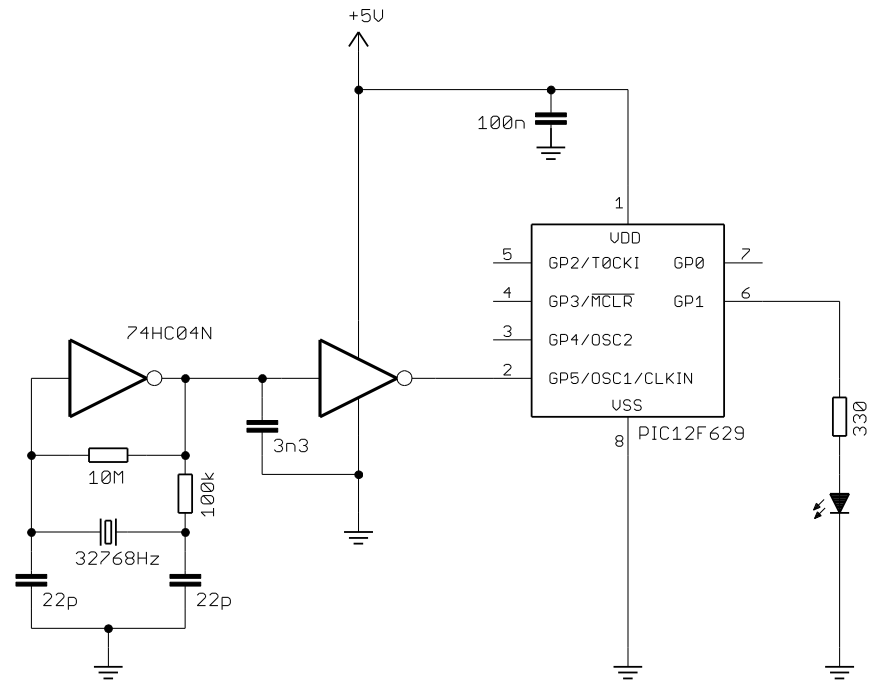
An external oscillator can be used as the PIC's clock source.

This is sometimes done so that the timing of various parts of a circuit is synchronised to the same clock signal. Or, your circuit may have an existing clock signal available, and it may make sense to use it if it is more accurate and/or stable than the PIC's internal RC oscillator – assuming you can afford the loss of one of the PIC's I/O pins.

To demonstrate the use of an external clock signal, we'll use a 32.768 kHz crystal oscillator, such as the one from [baseline assembler lesson 5](#), as shown in the circuit on the right.

To use an external oscillator with the PIC12F629, the 'EC' oscillator mode should be used, with the clock signal (with a frequency of up to 20 MHz) connected to the CLKIN input: pin 2 on a PIC12F629.

To implement this circuit using the [Gooligum training board](#), place a shunt in position 4 ("EC") of jumper block JP20, connecting the 32.768 kHz signal to CLKIN, and in JP3 and JP12 to enable the external $\overline{\text{MCLR}}$ pull-up resistor (not shown here) and the LED on GP1.



Since CLKIN uses the same pin as GP5, GP5 cannot be used for I/O when the PIC is in 'FOSC_EC' mode.

To illustrate the operation of this circuit, we can modify the crystal-driven LED flasher program developed in [lesson 2](#). In that example, the external 32.768 kHz signal was used to drive the Timer0 counter.

Now, however, the 32.768 kHz signal is driving the processor clock, giving an instruction clock rate of 8192 Hz. If Timer0 is configured in timer mode with a 1:32 prescale ratio, TMR0<7> will cycle at exactly 1 Hz (since $8192 = 32 \times 256$) – as is assumed in the example from [lesson 2](#).

Therefore, to adapt that program for this circuit, all we need to do is to change the configuration statement to:

```
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, external clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_EC);
```

and change the initialisation code from:

```
// configure Timer0
OPTION_REGbits.T0CS = 1;           // select counter mode
OPTION_REGbits.PSA = 0;           // assign prescaler to Timer0
OPTION_REGbits.PS = 0b110;        // prescale = 128
// -> incr at 256 Hz with 32.768 kHz input
```

to:

```
// configure Timer0
OPTION_REGbits.T0CS = 0;         // select timer mode
OPTION_REGbits.PSA = 0;           // assign prescaler to Timer0
OPTION_REGbits.PS = 0b100;       // prescale = 32
// -> incr at 256 Hz with 8192 Hz inst clock
```


With these changes made, the LED on GP1 should flash at almost exactly 1 Hz – to within the accuracy of the crystal oscillator.

Complete program

Here is the program from [lesson 2](#), modified as described above:

```

/*****
*
* Description: Lesson 5, example 2
*
* Demonstrates use of external clock mode
* (using 32.768 kHz clock source)
*
* LED flashes at 1 Hz (50% duty cycle),
* with timing derived from 8192 Hz instruction clock
*
*****/
*
* Pin assignments:
; GP1 = flashing LED
; CLKIN = 32.768 kHz signal
*
*****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, external clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_EC);

// Pin assignments
#define sFLASH sGPIO.GP1 // flashing LED (shadow)

/***** GLOBAL VARIABLES *****/
union { // shadow copy of GPIO
    uint8_t port;
    struct {
        unsigned GP0 : 1;
        unsigned GP1 : 1;
        unsigned GP2 : 1;
        unsigned GP3 : 1;
        unsigned GP4 : 1;
        unsigned GP5 : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation ***/

    // configure port
    TRISIO = ~(1<<1); // configure GP1 (only) as an output

    // configure Timer0

```

```

OPTION_REGbits.T0CS = 0;           // select timer mode
OPTION_REGbits.PSA = 0;           // assign prescaler to Timer0
OPTION_REGbits.PS = 0b100;       // prescale = 32
                                   // -> incr at 256 Hz with 8192 Hz inst clock

/** Main loop */
for (;;)
{
    // TMR0<7> cycles at 1 Hz, so continually copy to LED
    sFLASH = (TMR0 & 1<<7) != 0;   // sFLASH = TMR0<7>

    GPIO = sGPIO.port;             // copy shadow to GPIO
}

```

Crystals and ceramic resonators

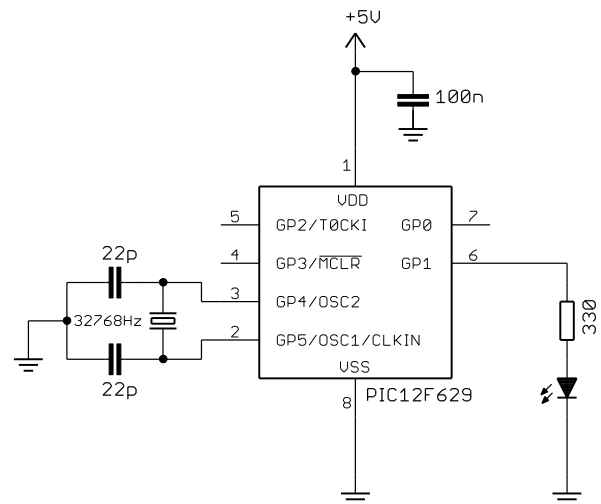
Generally, there is no need to build your own crystal oscillator; PICs include an oscillator circuit designed to drive crystals directly.

A parallel (not serial) cut crystal, or a ceramic resonator, is placed between the OSC1 and OSC2 pins, which are grounded via loading capacitors, as shown in the circuit diagram on the right.

You should consult the crystal or resonator manufacturer's data when selecting load crystals; those shown here are appropriate for a crystal designed for a load capacitance of 12.5 pF.

For some crystals it may be necessary to reduce the drive current by placing a resistor between OSC2 and the crystal, but in most cases it is not needed, and the circuit shown here (with the reset switch and pull-up omitted for clarity) can be used.

If you are using the [Gooligum training board](#), place shunts in position 2 ("32kHz") of JP20² and position 2 of JP21 ("32kHz"), connecting the 32.768 kHz crystal between OSC1 and OSC2, and close JP3 and JP12 to enable the external $\overline{\text{MCLR}}$ pull-up resistor (not shown here) and the LED on GP1.



The PIC12F629 offers three crystal oscillator modes: 'XT', 'LP' and 'HS'. They differ in the gain and frequency response of the drive circuitry.

'XT' ("crystal") is the mode most commonly used for crystals or ceramic resonators operating between 100 kHz and 4 MHz.

² You will find, with the Gooligum training board that the LED in this 32.768 kHz crystal example will flash, even with no shunt installed in JP20! This is because, when configured in `_LP_OSC` mode, the OSC1 input is very sensitive, and picks up crosstalk from the external 32.768 kHz signal on the board. If you want to prevent this effect, you can dampen the external 32.768 kHz signal by loading it with a 100 Ω resistor, placed between pin 1 of the 16-pin expansion header and ground, via the breadboard. The external clock example will still work with this resistor in place, and this 32.768 kHz crystal example will only work with shunts in the "32kHz" positions of JP20 and JP21 – as we'd expect.

‘HS’ (“high speed”) mode provides higher gain and is typically used for crystals or ceramic resonators operating above 4 MHz, up to a maximum frequency (on the 12F629) of 20 MHz. The higher drive level means that a series resistor is more likely to be necessary in ‘HS’ oscillator mode.

Lower frequencies generally require lower gain. The ‘LP’ (“low power”) mode uses less power and is designed to drive common 32.786 kHz “watch” crystals, although it can also be used with other low-frequency crystals or resonators.

The circuit shown above can be used to operate the PIC12F629 at 32.768 kHz, giving low power consumption and an 8192 Hz instruction clock, which, as we have seen, is easily divided to create an accurate 1 Hz signal.

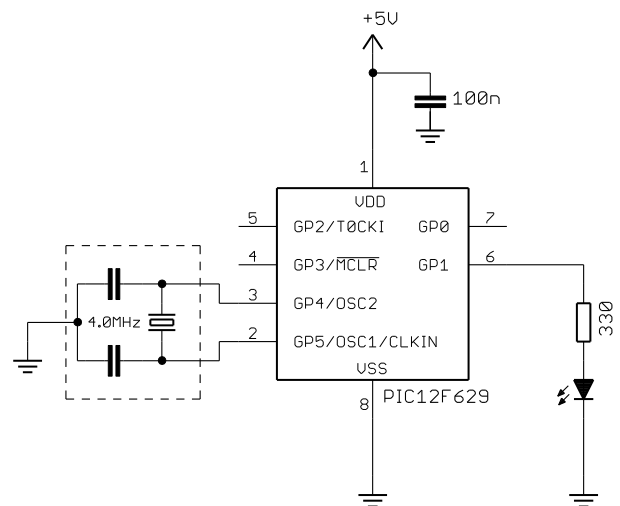
To flash the LED at 1 Hz, the program is exactly the same as for the external clock example above, except that the configuration statement must instead include the `FOSC_LP` option:

```
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, LP oscillator
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_LP);
```

Another option, when you want greater accuracy and stability than the internal RC oscillator can provide, but do not need as much as that offered by a crystal, is to use a ceramic resonator.

These are available in convenient 3-terminal packages which include appropriate loading capacitors, as shown in the circuit diagram (with the reset switch and pull-up omitted for clarity) on the right. The resonator package incorporates the components within the dashed lines.

If you have the [Gooligum training board](#), move the shunts to position 3 (“4MHz”) of JP20 and position 1 of JP21 (“4MHz”), connecting the 4.0 MHz resonator between OSC1 and OSC2, and leave JP3 and JP12 closed to enable the external `MCLR` pull-up resistor (not shown here) and the LED on GP1.



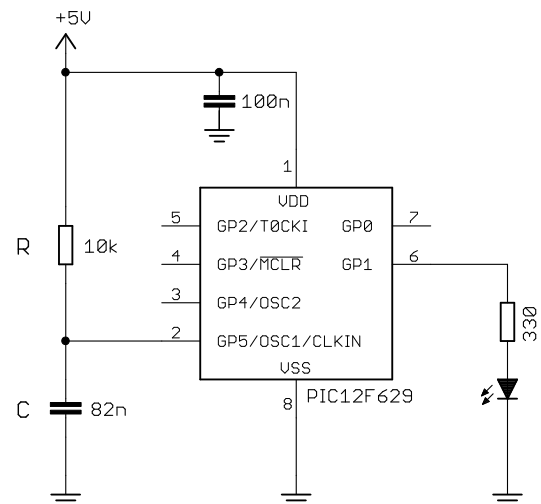
To test this circuit, you can change the ‘`FOSC_INTRCIO`’ configuration option to ‘`FOSC_XT`’ in the `__CONFIG()` macro in any program from the examples in any of the earlier lessons, since they all used a 4 MHz clock.

A good choice is the “flash an LED at exactly 1 Hz” program developed in [lesson 3](#), since it will generate an output of exactly 1 Hz, given a processor clock of exactly 4 MHz, and so should benefit from the more accurate clock source.

External RC oscillator

Finally, a low-cost, low-power option: mid-range PICs can use an oscillator based on an external resistor and capacitor, as shown (with the reset switch and pull-up omitted for clarity) on the right.

To implement this circuit using the [Gooligum training board](#), move the shunt to position 1 (“RC”) of JP20, connecting the 10 kΩ resistor and 82 nF capacitor to OSC1. Remove the shunt from JP21 and leave JP3 and JP12 closed, enabling the external $\overline{\text{MCLR}}$ pull-up resistor (not shown here) and the LED on GP1.



External RC oscillators, with appropriate values of R and C, can be useful when a very low clock rate is acceptable – drawing significantly less power than when the internal 4 MHz RC oscillator is used.

Running the PIC slowly can also simplify some programming tasks, needing fewer, shorter delays.

Microchip does not commit to a specific formula for the frequency (or period) of the external RC oscillator, only stating that it is a function of VDD, R, C and temperature, and in some documents providing some reference charts. But for rough design guidance, you can assume the period of oscillation is approximately $1.2 \times RC$.

Only use an external RC oscillator if the exact clock rate is unimportant.

Microchip recommends keeping R between 5 kΩ and 100 kΩ, and C above 20 pF.

In the circuit above, $R = 10 \text{ k}\Omega$ and $C = 82 \text{ nF}$.

Those values will give a period of approximately $1.2 \times 10 \times 10^3 \times 82 \times 10^{-9} \text{ s} = 984 \mu\text{s}$

Hence, we can expect to generate a clock frequency of around 1 kHz.

So, given a roughly 1 kHz clock, what can we do with it?

Flash an LED, of course!

Using a similar approach to before, we can use the instruction clock (approx. 256 Hz) to increment Timer0. In fact, with a prescale ratio of 1:256, TMR0 will increment at approx. 1 Hz.

TMR0<0> would then cycle at 0.5 Hz, TMR0<1> at 0.25 Hz, etc.

Now consider what happens when the prescale ratio is set to 1:64. TMR0 will increment at 4 Hz, TMR0<0> will cycle at 2 Hz, and TMR0<1> will cycle at 1 Hz, etc.

And that suggests a very simple way to make the LED on GP1 flash at 1 Hz:

If we continually copy TMR0 to GPIO, each bit of GPIO will reflect each corresponding bit of TMR0.

In particular, GPIO<1> will always be set to the same value as TMR0<1>. Since TMR0<1> is cycling at 1 Hz, GPIO<1> (and hence GP1) will also cycle at 1 Hz.

Complete program

The following program implements the approach described above. Note that the external RC oscillator is selected by using the option 'RCCLK' in the configuration statement.

```

/*****
*
* Description: Lesson 5, example 5
*
* Demonstrates use of external RC oscillator (~1 kHz)
*
* LED on GP1 flashes at approx 1 Hz (50% duty cycle),
* with timing derived from ~256 Hz instruction clock
*
*****/
*
* Pin assignments:
* GP1 = flashing LED
* OSC1 = R (10k) / C (82n)
*
*****/

#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, ext RC oscillator (~ 1kHz) + clkout
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_EXTRCCLK);

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation ***/

    // configure port
    TRISIO = ~(1<<1); // configure GP1 (only) as an output

    // configure Timer0
    OPTION_REGbits.T0CS = 0; // select timer mode
    OPTION_REGbits.PSA = 0; // assign prescaler to Timer0
    OPTION_REGbits.PS = 0b101; // prescale = 64
    // -> incr at 4 Hz with 256 Hz inst clock

    /*** Main loop ***/
    for (;;)
    {
        // TMR0<1> cycles at 1 Hz, so continually copy to LED (GP1)
        GPIO = TMR0; // copy TMR0 to GPIO
    }
}

```

The “main loop” is only a single assignment statement – by far the shortest “flash an LED” program we have done, demonstrating that slowing the clock rate can simplify certain programming problems. On the other hand, it is also the least accurate of the “flash an LED” programs, being only approximately 1 Hz. But for many applications, the exact speed doesn’t matter; it only matters that the LED visibly flashes, not how fast.

Power-On Reset

As explained in greater detail in [mid-range lesson 8](#), to reliably start program execution on a mid-range (or any) PIC, it is necessary to hold the device in a reset condition until the power supply has reached a consistently high enough voltage.

This was traditionally done by a simple RC circuit attached to the external $\overline{\text{MCLR}}$ pin. However, there is often no need to use external reset components with modern mid-range PICs, because they include a *power-up timer* (PWRT), which, if enabled, holds the device in reset for a nominal 72 ms from the initial *power-on reset* (POR) which occurs when power-on is detected.

The power-up timer is controlled by the $\overline{\text{PWRTE}}$ bit in the processor configuration word; setting $\overline{\text{PWRTE}}$ to '1' *disables* the power-on timer.

To enable it using XC8, include the symbol 'PWRTE_ON' in the `__CONFIG()` macro.

To disable it, use 'PWRTE_OFF' instead.

You may need to disable the power-up timer if your power supply takes more than 72 ms to settle. You should then use an external RC reset circuit, or an external supervisory circuit, such as one of Microchip's MCP10X devices, to hold the device in reset for longer. If so, it may be appropriate to disable the internal power-up timer, so that there is only one source of power-up delay.

But most of the time, unless your circuit is operating in difficult power supply conditions, you can enable the power-up timer (as we have done so far) and, if you are using an external reset, use a 10 k Ω resistor between $\overline{\text{MCLR}}$ and VDD.

If you are using the LP, XT or HS clock mode (which implies that you're probably using a crystal or resonator driven by the PIC's on-board oscillator circuitry), the *oscillator start-up timer* (OST) is invoked to give the crystal or resonator time to settle, after the PWRT delay completes. The OST counts pulses on the OSC1 pin, and holds the device in reset until it has counted 1024 oscillator cycles.

The OST is also used when the PIC wakes from sleep in LP, XT or HS clock mode, for the same reason – the oscillator is disabled while the device is in sleep mode, and takes a while to start and become stable.

Note that the OST is invoked whether or not PWRT is enabled. The only way to avoid the oscillator start-up delay is to use one of the EC, internal RC or external RC oscillator modes.

For fastest processor start-up at power-on, disable the power-up timer and use an external clock, avoiding both the PWRT and OST delays – and hope that you have a very fast-starting and stable power supply! But it's generally best to simply accept that your program won't start running for up to 100 ms after you turn the power on...

Brown-out Detect

[Mid-range lesson 8](#) also explained that the PIC's operation can become unreliable if the power supply voltage falls too far during normal operation – a condition known as a *brown-out*. In general, it is preferable to stop program execution while the brown-out situation persists, instead of risking unreliable operation; it's better to be able to recover cleanly after the brown-out, instead of not knowing what your program might do.

Most mid-range PICs provide a *brown-out detect* (BOD, also called *brown-out reset*, or BOR) facility, which, if enabled, will reset the device if the supply voltage falls below the brown-out detect voltage

(between 2.025 V and 2.175 V on the PIC12F629), and hold it in reset until the voltage rises again. If the power-up timer is enabled (recommended if you are using BOD), the device will remain in reset for a further 72 ms after the brown-out condition clears – and if another brown-out occurs during this PWRT delay, it will be detected and the process will repeat.

Brown-out detection on the PIC12F629 is controlled by the **BODEN** bit in the processor configuration word; setting **BODEN** to '1' enables brown-out detection.

To enable BOD (or BOR) using XC8, use the symbol 'BOREN_ON' in the `__CONFIG()` macro.

To disable it, use 'BOREN_OFF' instead.

Detecting a brown-out reset

If a brown-out occurs, resetting the PIC and hence restarting your program, you may want your application to react to this, behaving differently to a power-on, watchdog timer, or other reset. For example, if your program has restarted because of a brown-out, you may want it to try to continue doing whatever it was doing before the brown-out, instead of running through the full initialisation routine.

Fortunately, mid-range PICs provide flags which allow us to detect and respond differently to both power-on and brown-out resets.

In the 12F629, these flags are contained in the power control register, **PCON**.

The $\overline{\text{POR}}$ (power-on reset status) flag is cleared when a power-on reset occurs, and is set if a brown-out reset occurs. It is unaffected by all other resets.

This means that, to use this flag to differentiate power-on from other resets, you must set $\overline{\text{POR}}$ to '1' whenever a power-on reset occurs. Since all the other types of reset either set this bit or leave it unchanged, it will then only ever be '0' when a power-on reset has occurred.

Similarly, the $\overline{\text{BOD}}$ (brown-out detect status) flag is cleared when a brown-out reset occurs, and is unaffected by all other resets.

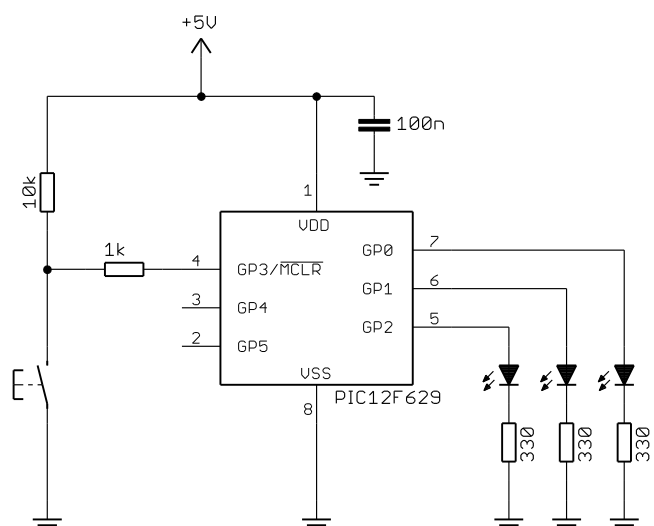
So to use this flag to differentiate brown-out from other resets, you must set $\overline{\text{BOD}}$ to '1' following power-on. Since all the other resets leave this bit unchanged, it will only ever be '0' when a brown-out has occurred.

Since $\overline{\text{BOD}}$ is unaffected by a power-on reset, its value is unknown when the device is first powered on. Therefore, the first flag you should test is $\overline{\text{POR}}$. If it is clear, you can be sure that a power-on reset has occurred, and you can then set both $\overline{\text{POR}}$ and $\overline{\text{BOD}}$, ready for testing after subsequent resets.

An example may help to clarify this.

We'll use the circuit shown on the right, which you can implement with the [Gooligum training board](#) by closing jumpers JP3, JP11, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP0, GP1 and GP2.

If you are using Microchip's Low Pin Count Demo Board, you can connect LEDs to GP0, GP1 and GP2, by making connections on the 14-pin header: 'RA0' to 'RC0', 'RA1' to 'RC1' and 'RA2' to 'RC2'.



The program will simply light the LED on GP0, regardless of why the PIC had been reset (or powered on). In addition, the LED on GP1 will be lit on power-on (and not for any other reset), and the LED on GP2 will indicate that a brown-out has occurred.

The pushbutton will be used to generate an external $\overline{\text{MCLR}}$ reset. When this happens, only the LED on GP0 should light, because the reset is caused by neither power-on nor brown-out.

After enabling brownout detection in the device configuration:

```
// ext reset, no code protect, brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_ON & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);
```

After initialising TRISIO and clearing GPIO (so that all LEDs are initially off), the first task is to test the $\overline{\text{POR}}$ flag to see if a power-on reset has occurred. If so, we should set the $\overline{\text{POR}}$ and $\overline{\text{BOD}}$ flags, to set them up for any subsequent resets (as discussed above), and light the POR LED:

```
if (!PCONbits.nPOR)           // if power-on reset (/POR = 0),
{
    PCONbits.nPOR = 1;        // set POR and BOD flags for next reset
    PCONbits.nBOD = 1;
    sP_LED = 1;              // enable POR LED (shadow)
}
```

A shadow copy of GPIO is used to avoid potential read-modify-write problems, as we have done [before](#).

Now we can reliably test for a brown-out reset, and, if one has occurred, set the $\overline{\text{BOD}}$ flag for next time, and light the BOD LED:

```
if (!PCONbits.nBOD)          // if brown-out detect (/BOD = 0)
{
    PCONbits.nBOD = 1;        // set BOD flag for next reset
    sB_LED = 1;              // enable BOD LED (shadow)
}
```

Note that, if a power-on reset had occurred, this brown-out detect code will never be executed, because the earlier code sets the $\overline{\text{BOD}}$ flag, whenever a power-on reset is detected.

Finally, regardless of the reason for the reset, we light the “on” LED and copy the shadow register to the port:

```
// enable "on" indicator LED
sO_LED = 1;                  // (via shadow register)

// light enabled LEDs
GPIO = sGPIO.port;          // copy shadow GPIO to port
```

If the pushbutton is pressed, generating a $\overline{\text{MCLR}}$ reset, only this “on” LED will be lit.

Finally, we simply wait until the next reset:

```
for (;;)                    // wait forever
;
```


Complete program

Here is how these pieces fit together:

```

/*****
*
* Description: Lesson 5, example 6
*
* Demonstrates use of brown-out detect
* and differentiation between POR, BOD and MCLR resets
*
* Turns on POR LED only if power-on reset is detected
* Turns on BOD LED only if brown-out detect reset is detected
* Turns on indicator LED in all cases
* (no POR or BOD implies MCLR, as no other reset sources are active)
*
*****/
*
* Pin assignments:
* GP0 = "on" indicator LED (always turned on)
* GP1 = POR LED (indicates power-on reset)
* GP2 = BOD LED (indicates brown-out detected)
*
*****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_ON & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sO_LED  sGPIO.GP0  // "on" indicator LED - always on (shadow)
#define sP_LED  sGPIO.GP1  // POR LED to indicate power-on reset (shadow)
#define sB_LED  sGPIO.GP2  // BOD LED to indicate brown-out (shadow)

/***** GLOBAL VARIABLES *****/
union {
    uint8_t      port;           // shadow copy of GPIO
    struct {
        unsigned GP0    : 1;
        unsigned GP1    : 1;
        unsigned GP2    : 1;
        unsigned GP3    : 1;
        unsigned GP4    : 1;
        unsigned GP5    : 1;
    };
} sGPIO;

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation ***/

    // configure port
    GPIO = 0;           // start with all LEDs off
}

```

```

sGPIO.port = 0;           // update shadow
TRISIO = 0b111000;       // configure GP0, GP1 and GP2 as outputs

// check for POR or BOD reset
if (!PCONbits.nPOR)      // if power-on reset (/POR = 0),
{
    PCONbits.nPOR = 1;    // set POR and BOD flags for next reset
    PCONbits.nBOD = 1;
    sP_LED = 1;          // enable POR LED (shadow)
}
if (!PCONbits.nBOD)     // if brown-out detect (/BOD = 0)
{
    PCONbits.nBOD = 1;    // set BOD flag for next reset
    sB_LED = 1;          // enable BOD LED (shadow)
}

/** Main code */

// enable "on" indicator LED
sO_LED = 1;              // (via shadow register)

// light enabled LEDs
GPIO = sGPIO.port;      // copy shadow GPIO to port

// wait forever
for (;;)
    ;
}

```

To test this program, you will need a variable power supply.

If you have the [Gooligum training board](#), you can connect your power supply to Vdd and ground via pins 15 ('+V') and 16 ('GND') on the 16-pin expansion header.

You should find that if you set the supply to say 4 V and apply power, the POR LED (GP1) should light, along with the "on" LED (GP0)

If you then simulate a brown-out, by lowering the voltage until both LEDs turn off (at around 2 V; by this time they will be very dim, since the forward voltage of most normal-brightness LEDs is around 2 V), without taking the voltage all the way to zero, and then raise the voltage again, the BOD LED (GP2) should light, indicating that the brown-out was detected. The "on" LED should light, as always, but not POR, because this was a brown-out, not a power-on reset..

If you then turn off the power supply, and turn it back on again, the POR LED should light again, and not BOD, because this was a normal power-on, not a brown-out.

Finally, if you press the pushbutton, generating a $\overline{\text{MCLR}}$ reset, while either the POR or BOD LED is lit, all the LEDs will go out while the button is pressed, and then only the "on" LED will come on, indicating that this reset was neither a power-on nor a brown-out.

Summary

Most of the examples in this lesson did not require any new programming techniques; the first few being minor adaptations of programs from earlier lessons, with different processor configuration options, to select the oscillator mode being demonstrated.

However, the final example demonstrated that power-on and brown-out resets can be detected and responded to effectively, using the XC8 compiler – the detection code being simple and elegant, compared with the assembler version.

The [next lesson](#) will revisit material from [mid-range assembler lessons 9](#) and [11](#), focussing on comparators – the single comparator in the PIC12F629, and the dual comparator module in the PIC16F684.

Migrating to Enhanced Mid-Range PICs

Enhanced Mid-Range Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 1: Basic Digital I/O

This series of lessons introduces the enhanced mid-range PIC architecture, using assembly language, and follows on from the [Mid-range PIC Assembler](#) tutorial series. It assumes that you are familiar with the content of at least the free mid-range lessons (1-8).

Although the mid-range architecture was a step up from the baseline architecture, it still has a number of limitations that we've had to work around, such as the read-modify-write problem and the need to address data memory in contiguous blocks of no more than 96 bytes.

The enhanced mid-range (12F1xxx and 16F1xxx) PIC architecture overcomes these limitations and more, removing the need to use shadow registers, extending the maximum code and data memory sizes, making this memory more easily addressable, interrupts easier to use, the stack larger and its contents accessible, and includes a number of useful additional assembly language instructions, as we'll see.

This lesson introduces one of the simplest of the enhanced mid-range PICs – the PIC12F1501. It then goes on to describe basic digital I/O output, as covered in lessons [1](#) to [3](#) of the mid-range assembler tutorial series.

In summary, this lesson covers:

- Introduction to the PIC12F1501
- Simple digital input and output
- Selecting the internal oscillator frequency
- Using internal (weak) pull-ups
- Banking and paging

Getting Started

As before, these tutorials assume that you are using either the [Gooligum Baseline and Mid-range PIC Training and Development Board](#) or Microchip's Low Pin Count Demo Board¹, with Microchip's MPLAB 8 or MPLAB X integrated development environment.

You will also need a programmer, such as Microchip's PICkit 3, which is compatible the enhanced mid-range PICs.

*Note: the PICkit 2 programmer, which was used in the baseline and mid-range tutorials, is **not supported** in MPLAB for use with enhanced mid-range PICs.*

¹ it is of course possible to adapt these lessons to different development boards

Introducing the PIC12F1501

The 12F1501 is in some ways the simplest of the enhanced mid-range PICs².

It is roughly equivalent to the 12F675, which in turn is essentially the same as the 12F629 introduced in [mid-range lesson 1](#), with the addition of an analog-to-digital converter (ADC) – as can be seen in the following table, which summarises some of the basic features of various 8-pin PICs:

Device	Memory (words or bytes)			Timers		Analog		Clock rate (max MHz)
	Program	Data	EEPROM	8-bit	16-bit	Comp-arators	ADC inputs	
12F508	512	25	0	1	0	0	0	4
12F629	1024	64	128	1	1	1	0	20
12F675	1024	64	128	1	1	1	4	20
12F683	2048	128	256	2	1	1	4	20
12F1501	1024	64	0	2	1	1	4	20
12F1822	2048	128	256	2	1	1	4	32
12F1840	4096	256	256	2	1	1	4	32

However, the 12F1501 also includes a number of peripherals which the 12F675 does not, such as a digital-to-analog converter (DAC), pulse-width modulation (PWM) outputs, a configurable logic cell (used to implement simple logic functions in hardware) and a numerically controlled oscillator.

And although the 12F1501 does not include any EEPROM memory, it has the ability to write into its program (Flash) memory, 128 bytes of which is “high-endurance” Flash which can be re-written at least 100,000 times – making it a viable alternative to true EEPROM memory in many situations.

We’ll explore these additional features in later lessons.

As explained in [mid-range lesson 1](#), the data memory in mid-range PICs consists of up to four banks, with 128 addresses in each bank. The first 32 addresses can hold special function registers (SFRs), used for core functionality and to access peripherals such as timers. The remaining 96 addresses in each bank are available for general-purpose registers (GPRs), used for temporary data storage such as program variables. Some of these SFRs and GPRs are mapped into every bank, so that they can be accessed regardless of which bank is currently selected.

In the enhanced mid-range PIC architecture, this scheme is greatly expanded, with every enhanced mid-range device having 32 banks of 128 addresses each.

Each bank (other than bank 31, as we’ll see later) is laid out the same way, as illustrated on the right.

Standard bank layout

Offset	Register Type
00h	Core Registers
0Bh	
0Ch	
1Fh	Special Function Registers
20h	
6Fh	
70h	General Purpose RAM
7Fh	
7Fh	Common RAM

² which is why we’re starting this series with the PIC12F1501...

The first 12 addresses of every bank provide access to the “core registers”, as shown on the right.

There is only one set of core registers, but they appear in the same locations within each bank – meaning that they can be accessed in the same way, whichever bank is selected.

You’ll recognise some of these: **STATUS**, **PCL**, **PCLATH** and **INTCON** are all carried over from the mid-range architecture, where they were also mapped into every bank, and operate the same way as before.

One difference however is that **STATUS** no longer contains any bank selection bits. Now that we have 32 ($= 2^5$) banks, we need five bits to specify which bank is selected. These bits are now held in a dedicated bank selection register: **BSR**.

WREG is our old friend **W**, the working register, now mapped into data memory. This means that any instruction which accesses data memory can now work directly on **W**.

For example, it is now possible to rotate **W** directly (something that hadn’t been possible before), with, for example:

```
rlf    WREG, w
```

will rotate the value in **W** left (through carry)³, writing the result back into **W**.

The remaining core registers (**INDF0**, **INDF1**, **FSR0L**, **FSR0H**, **FSR1L** and **FSR1H**) are used for indirect addressing, in a similar, but greatly enhanced, way to the mid-range architecture’s **INDF** and **FSR** registers, as described in mid-range lesson 14.

In fact, indirect addressing is one of the most significant enhancements in the enhanced mid-range architecture. The new indirect addressing scheme allows for both data and program memory to be read and written. And it allows for general purpose RAM to be accessed linearly, making it easy to implement arrays and other large structures, without being constrained by bank boundaries. We’ll look in detail at how it works in a later lesson.

The last 16 addresses of each bank are used for “common RAM” – this is equivalent to the “shared” data memory we’re familiar with from baseline and mid-range PICs⁴. It contains 16 GPRs which are all mapped into the same location in every bank – avoiding the need for bank selection instructions when accessing data stored in common RAM.

Of course we often need more than 16 bytes of data memory, whether for variables, buffers or whatever – which is why the enhanced mid-range architecture also provides “general purpose RAM”, consisting of up to 80 GPRs in each of banks 0 to 30. These registers are banked – each only appears in a single bank. This means that enhanced mid-range PICs can have up to $80 \times 31 = 2480$ bytes of general purpose RAM.

The 12F1501, however, has only 48 bytes of general purpose RAM, all in bank 0, which along with the 16 bytes of common RAM (mapped into every bank) give it a total of 64 bytes of data memory.

Core Registers

Offset	Register Name
00h	INDF0
01h	INDF1
02h	PCL
03h	STATUS
04h	FSR0L
05h	FSR0H
06h	FSR1L
07h	FSR1H
08h	BSR
09h	WREG
0Ah	PCLATH
0Bh	INTCON

³ see [baseline lesson 11](#) for an explanation of the `rlf` instruction

⁴ introduced in [baseline lesson 3](#)

The remaining 20 addresses in banks 0 to 30 hold the special function registers used to access the device's ports and peripherals. With $20 \times 31 = 620$ addresses available for SFRs, the enhanced mid-range architecture has enough space in its register map to support a much wider range of advanced peripherals than was possible in mid-range PICs.

But since the 12F1501 is a relatively basic device, it doesn't need hundreds of SFRs. Many of its 32 banks are empty, other than the core registers and common RAM, which are mapped into every bank.

It doesn't make sense to list them all here – you should refer to the PIC12F1501 data sheet for that – but here are the first four banks, where most of the commonly-used SFRs are located:

PIC12F1501 Special Function Registers (banks 0 – 3)

<i>Bank 0</i>		<i>Bank 1</i>		<i>Bank 2</i>		<i>Bank 3</i>	
000h	Core Registers	080h	Core Registers	100h	Core Registers	180h	Core Registers
00Bh		08Bh		10Bh		18Bh	
00Ch	PORTA	08Ch	TRISA	10Ch	LATA	18Ch	ANSELA
00Dh		08Dh		10Dh		18Dh	
00Eh		08Eh		10Eh		18Eh	
00Fh		08Fh		10Fh		18Fh	
010h		090h		110h		190h	
011h	PIR1	091h	PIE1	111h	CM1CON0	191h	PMADRL
012h	PIR2	092h	PIE2	112h	CM1CON1	192h	PMADRH
013h	PIR3	093h	PIE3	113h		193h	PMDATL
014h		094h		114h		194h	PMDATH
015h	TMR0	095h	OPTION_REG	115h	CMOUT	195h	PMCON1
016h	TMR1L	096h	PCON	116h	BORCON	196h	PMCON2
017h	TMR1H	097h	WDTCON	117h	FVRCON	197h	VREGCON
018h	T1CON	098h		118h	DACCON0	198h	
019h	TMR2	099h	OSCCON	119h	DACCON1	199h	
01Ah	PR2	09Ah	OSCSTAT	11Ah		19Ah	
01Bh	T2CON	09Bh	ADRESL	11Bh		19Bh	
01Ch		09Ch	ADRESH	11Ch		19Ch	
01Dh		09Dh	ADCON0	11Dh	APFCON	19Dh	
01Eh		09Eh	ADCON1	11Eh		19Eh	
01Fh		09Fh	ADCON2	11Fh		19Fh	
020h	General Purpose RAM	0A0h		120h		1A0h	
04Fh							
050h							
06Fh		0EFh		16Fh		1EFh	
070h	Common RAM	0F0h	Common RAM	170h	Common RAM	1F0h	Common RAM
07Fh		0FFh		17Fh		1FFh	

Many of these will be familiar from the mid-range PICs – or at least similar enough that you can guess what they are for (such as three PIR registers, compared with one on the PIC16F684, and three ADCON

registers instead of two). Others are associated with new functionality or peripherals, which we'll look at in future lessons, instead of going into detail now.

Finally, bank 31 provides enhanced functionality related to interrupts and the hardware stack.

We'll look at some of these features in more detail in later lessons, but briefly – we saw in [mid-range lesson 6](#) that, to avoid interfering with the main program, the interrupt service routine must save and restore the processor *context* (the content of registers such as *W* and *STATUS*). Enhanced mid-range devices do this for us, with most of the core registers being automatically saved into bank 31 when an interrupt occurs, and then automatically restored when the interrupt exits.

The stack is now 16 registers deep (compared with 8 in mid-range PICs), and although it's not something you need to do often (if ever) the stack contents can now be accessed via a set of registers in bank 31.

As mentioned above, bank selection in the enhanced mid-range architecture is done by writing into bank number into the bank selection register, *BSR*. Bank selection is something that is done so often that a dedicated instruction is now provided specifically for this:

'`movlb k`' loads the literal value 'k' into the bank selection register – “**move literal to BSR**”

Thus, to select bank 2, you could write:

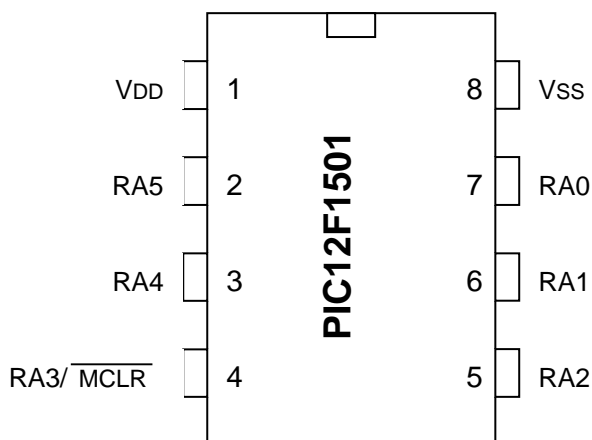
```
movlb 2 ; select bank 2
```

This makes it possible to select any bank with a single instruction, in a single instruction clock cycle, without affecting *W* or any *STATUS* flags. That's a step forward from the mid-range architecture, which needs two bit set or clear instructions on four-bank devices such as the PIC16F887.

But as explained in [baseline lesson 3](#), it is better to use the `banksel` assembler directive, which will generate the appropriate `movlb` instruction to select the bank corresponding to the specified register address. As in the mid-range architecture, some SFRs are grouped, so that once the correct bank is selected for one of them, you can be sure that the bank selection will not need to be changed before accessing other registers in the group. But if you are ever in doubt, use `banksel`.

PIC12F1501 Input / Output

Like the 12F629, the 12F1501 provides six I/O pins in an eight-pin package:



VDD is the positive power supply and VSS is the “negative” supply, or ground.

VDD (relative to VSS) on the PIC12F1501 can range from 2.3 V to 5.5 V, although at least 2.5 V is needed if the clock rate is greater than 16 MHz⁵.

⁵ A low-power variant, the PIC12LF1501, is also available, where VDD can range from 1.8 V to 3.6 V, with at least 2.5 V needed for clock rates above 16 MHz.

The remaining pins, RA0 to RA5, are the I/O pins. Just like the baseline and mid-range 8-pin PICs, they are used for digital input and output, except for RA3, which can only be used as an input. The other pins can be individually set to be inputs or outputs.

Each of these I/O pins has more functions that can be assigned to it – too many to show on the diagram above. As we’ll see in a later lesson, some of these functions can be selectively mapped to alternate pins. And as usual, some functions may need to be disabled before a pin can be used for digital I/O.

Taken together, the six I/O pins comprise the general-purpose I/O *port*, or GPIO port, which is referred to as PORTA on the 12F1501.

As in the midrange architecture, there is a TRIS register (TRISA) associated with the port which controls whether each pin is set as an input or output:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TRISA			TRISA5	TRISA4		TRISA2	TRISA1	TRISA0

It works the same way as the TRIS registers we’ve seen before, with a ‘1’ configuring the corresponding pin as an input, and a ‘0’ configuring it as an output. And as usual, every pin is configured as an input by default; TRISA is set to all ‘1’s when the device is powered on.

As you’d expect, there is also a “PORTA” register which provides access to the port pins:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTA			RA5	RA4	RA3	RA2	RA1	RA0

If a pin is configured as an output, setting the corresponding PORTA bit to ‘1’ outputs a ‘high’ on that pin; clearing it to ‘0’ outputs a ‘low’.

Reading the PORTA register reads the voltage present on each pin, assuming that the pins being read are configured for digital I/O⁶. If the voltage on a pin is high, the corresponding bit reads as ‘1’; if the input pin is low, the corresponding bit reads as ‘0’.

Note: the port registers represent the actual voltages present on each digital I/O pin, including pins configured as digital outputs

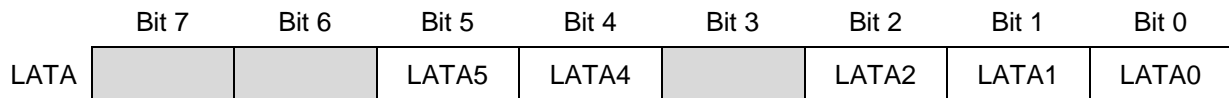
This behaviour is the same as in the baseline and mid-range PIC architectures. If you attempt to output a ‘high’ on a pin by writing a ‘1’ to the corresponding port bit, but the external circuit holds that pin low, that pin will read as ‘0’ – not what you might have expected.

As was explained in [baseline lesson 2](#), this behaviour can lead to what are known as *read-modify-write* problems, where instructions which are intended to modify only specific pins actually read the entire port, including pins which may not reflect the value that had been output to them, and then write the new value (with some bits possibly incorrect) back to the port.

To work around this potential problem, we have been using *shadow registers*, to keep track of what we have output to the port pins, and to operate on that copy, periodically writing it out to the port.

⁶ that is, any functionality on a pin which interferes with digital I/O operation, such as being configured as an analog input (see for example [baseline lesson 10](#)), has been disabled

To avoid read-modify-write problems, the enhanced mid-range architecture makes available an “output data latch” register, associated with each port:



Writing to LATA has the same effect as writing to PORTA: if a pin is configured as an output, setting the corresponding LATA bit to ‘1’ outputs a ‘high’ on that pin; clearing it to ‘0’ outputs a ‘low’.

However – reading LATA returns the value that was last written to LATA. It does not read the voltages on the pins (whether input or output) themselves.

This means that **in the enhanced mid-range architecture, there is no need to use shadow registers**, as long as you follow these rules:

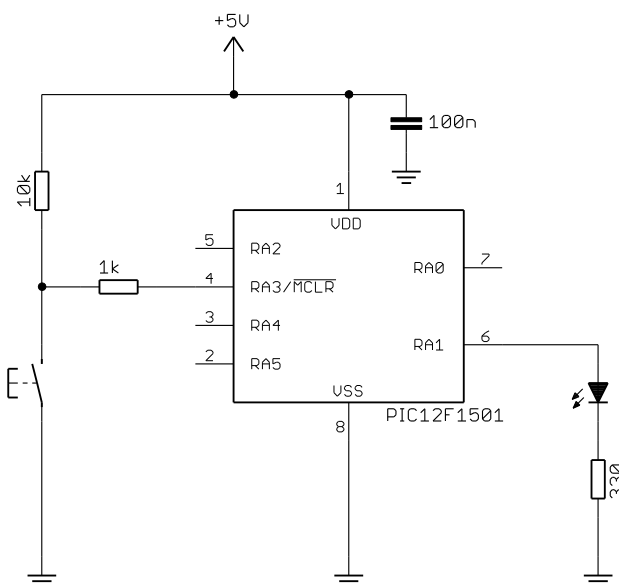
- if you are writing an entire byte to a port, you can write to either PORTA or LATA
- if you are modifying individual port pins, you should operate on LATA
- if you are reading digital input pins, you must read PORTA

To keep it simpler, you won’t run into any problems if you always access LATA to write to or modify output pins, and PORTA to read digital input pins.

Example 1: Turning on an LED

We’ll start the same way that we did in [mid-range lesson 1](#), by simply lighting a single LED, connected to one of the 12F1501’s digital I/O pins.

And we’ll use the same circuit as before:



In the same way as the 8-pin baseline and mid-range PICs, pin 4 can be configured as either a digital input (RA3) or as an external reset (“master clear”, $\overline{\text{MCLR}}$), which, if pulled low, will reset the processor.

In this example, we’ll configure the PIC for external reset, allowing either the pushbutton or PICKit 3 to pull $\overline{\text{MCLR}}$ low, resetting the device.

If you are using the [Gooligum training board](#), plug the PIC12F1501 into the top section of the 14-pin IC socket – the section marked ‘12F’. Close jumpers JP3 and JP12 to bring the 10 k Ω resistor into the circuit and to connect the LED to RA1, and ensure that every other jumper is disconnected.

With this simple circuit in place, and connected to your PC via a suitable programmer, such as a PICKit 3, it’s time to move on to programming!

As usual, after the comment block at the start of the program, we `#include` the processor include file:

```
#include "p12F1501.inc"
```

Note that the ‘`list p=`’ directive, specifying the processor, is not included in the templates provided with MPLAB X and is no longer recommended for normal use, so we will no longer use it in these tutorials.

Next the processor is configured:

```
;***** CONFIGURATION
        ; ext reset, internal oscillator (no clock out), no watchdog,
        ;   brownout resets on, no power-up timer, no code protect
        ; no write protection, stack resets on, low brownout voltage,
        ;   no low-power brownout reset, high-voltage programming
__CONFIG __CONFIG1, _MCLRE_ON & _FOSC_INTOSC & _CLKOUTEN_OFF & _WDTE_OFF &
_BOREN_ON & _PWRTE_OFF & _CP_OFF
__CONFIG __CONFIG2, _WRT_OFF & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF
```

[each `__CONFIG` directive must be written as a single line in the assembler source code]

The 12F1501 has too many configuration options to fit into a single 14-bit word, so it has two configuration words, each defined via a `__CONFIG` directive as we've done before, but with an additional parameter, `'_CONFIG1'` or `'_CONFIG2'` specifying which configuration word is being defined, as shown.

We've seen many of these configuration options before, and those we haven't we'll examine in greater detail in later lessons, but briefly the options being selected here are:

- `_MCLRE_ON`
Enables the external reset, or “master clear” ($\overline{\text{MCLR}}$) on pin 4.
- `_FOSC_INTOSC`
Selects the internal RC oscillator as the clock source
- `_CLKOUTEN_OFF`
Disables the clock out function on pin 3
- `_WDTE_OFF`
Disables the watchdog timer.
- `_BOREN_ON`
Enables brown-out reset.
- `_PWRTE_OFF`
Disables the power-up timer.
- `_CP_OFF`
Turns off program memory code protection.
- `_WRT_OFF`
Disables flash memory write protection.
- `_STVREN_ON`
Enables stack overflow/underflow resets. With this enabled, the processor will reset if you cause a stack *overflow* by (for example) nesting subroutine calls too deeply, or an *underflow* by (for example) mistakenly executing a `return` instruction outside a subroutine.
- `_BORV_LO`
Selects the low brown-out reset voltage option
- `_LPBOR_OFF`
Disables the low-power brown-out reset facility.
- `_LVP_OFF`
Disables low-voltage programming.

To program the device, a high voltage (around 12 V) must be applied to the VPP pin, as was the case with the baseline and mid-range PICs we've examined. Low-voltage programming mode avoids the need for this high voltage, but we don't need it because the PICkit 3 can operate in the “normal” high-voltage programming mode.

Note that, like the PIC16F684, there is no need to load a value to calibrate the internal RC oscillator in the 12F1501 – nor is there any facility for doing so.

As in the mid-range architecture, the *reset vector*, where program execution begins, is always at the start of memory: address 0000h.

So we should start our main code section with:

```
;***** RESET VECTOR *****
RES_VECT CODE    0x0000          ; processor reset vector
```

Note that the section is labelled ‘RES_VECT’, instead of ‘RESET’, which we’d used in the mid-range lessons. This is necessary because the enhanced mid-range architecture includes a new instruction, ‘reset’, which as you might expect performs a software reset operation, allowing your program to restart itself cleanly.

Next we configure the RA1 pin as an output, as we’ve done many times before:

```
; configure port
movlw    ~(1<<RA1)          ; configure RA1 (only) as an output
banksel TRISA
movwf   TRISA
```

To make RA1 output a ‘high’, we have to set bit 1 of PORTA to ‘1’, which we could do with:

```
; turn on LED
banksel PORTA
movlw    1<<RA1            ; set RA1 high
movwf   PORTA
```

because there is no risk of running into read-modify-write problems when writing an entire byte, updating the port register in a single operation, like this.

However, it is better to get into the habit of writing to LATA to modify output pins.

In particular, by using LATA, we can safely use instructions such as `bsf`, which operate by reading the register, modifying one or more bits (setting a single bit in this case), and then writing the result back to the port, i.e. a read-modify-write operation.

So on an enhanced mid-range device, we can safely use:

```
; turn on LED
banksel LATA
bsf     LATA,RA1           ; set RA1 high
```

Finally, we need an “infinite loop”, to stop the program running off into the rest of (uninitialised) program memory:

```
; loop forever
goto   $
```

```
END
```

And of course an ‘END’ directive has to go at the end of the source code.

If you assemble these pieces of, the assembler will give you a couple of messages like:

```
Message[302] C:\...\EA_L1-TURN_ON_LED.ASM 64 : Register in operand not in bank
0. Ensure that bank bits are correct.
Message[303] C:\...\EA_L1-TURN_ON_LED.ASM 70 : Program word too large.
Truncated to core size. (F9C4)
```

We've seen the [302] message, warning that the code references a register which is not in bank 0, previously in the mid-range assembler lessons.

The [303] message is new. It's generated because the processor configuration constants are defined in the 'p12f1501.inc' include file as 16-bit values, while the 12F1501's configuration words are only 14 bits wide. 16 bit into 14 don't go, so the assembler has to truncate the config values to 14 bits by dropping the top two bits.

We don't need to see these warnings, so it's ok to disable them, using 'errorlevel' directives:

```
errorlevel -302          ; no warnings about registers not in bank 0
errorlevel -303          ; no warnings about program word too large
```

These should be placed toward the beginning of your program, as usual.

Complete program

Putting together all the above, here's our complete assembler source for turning on an LED:

```

;*****
;
; Description:      Migration lesson 1, example 1
;
; Turns on LED.   LED remains on until power is removed.
;
;*****
;
; Pin assignments:
;   RA1 = indicator LED
;
;*****
#include "p12F1501.inc"

errorlevel -302          ; no warnings about registers not in bank 0
errorlevel -303          ; no warnings about program word too large

;***** CONFIGURATION
;   ext reset, internal oscillator (no clock out), no watchdog,
;   brownout resets on, no power-up timer, no code protect
;   no write protection, stack resets on, low brownout voltage,
;   no low-power brownout reset, high-voltage programming
__CONFIG __CONFIG1, _MCLRE_ON & _FOSC_INTOSC & _CLKOUTEN_OFF & _WDTE_OFF &
_BOREN_ON & _PWRTE_OFF & _CP_OFF
__CONFIG __CONFIG2, _WRT_OFF & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF

;***** RESET VECTOR *****
RES_VECT CODE    0x0000          ; processor reset vector

;***** MAIN PROGRAM *****

```

```

;***** Initialisation
start
    ; configure port
    movlw    ~(1<<RA1)        ; configure RA1 (only) as an output
    banksel TRISA
    movwf   TRISA

;***** Main code
    ; turn on LED
    banksel LATA
    bsf     LATA,RA1        ; set RA1 high

    ; loop forever
    goto   $

END

```

Example 2: Flashing an LED (50% duty cycle)

Having lit a single LED, the next step is to make it flash (as always...).

Although it usually makes more sense to use a timer (see for example [mid-range lesson 4](#), or, using interrupts, [mid-range lesson 6](#)) to time a process such as flashing an LED, you might occasionally want to use delay loops, as we did in [mid-range lesson 1](#), where we toggled an LED every 500 ms.

That lesson included a routine which used a couple of nested loops to do nothing but loop around wasting time for approximately 500,000 instruction cycles, as follows:

```

    ; delay 500 ms
    movlw    .244            ; outer loop: 244 x (1023 + 1023 + 3) + 2
    movwf   dc2              ; = 499,958 cycles
    clrf    dc1              ; inner loop: 256 x 4 - 1
dly1      nop                ; inner loop 1 = 1023 cycles
    decfsz  dc1,f
    goto    dly1
dly2      nop                ; inner loop 2 = 1023 cycles
    decfsz  dc1,f
    goto    dly2
    decfsz  dc2,f
    goto    dly1

```

This code assumes a 4 MHz processor clock, which implies a 1 MHz instruction clock, or 1 μ s per instruction cycle. 500,000 instruction cycles would then equate to 500 ms.

The PIC12F629 used in that lesson has an internal RC oscillator which can only provide a fixed 4 MHz processor.

However, the 12F1501's internal RC oscillators can be configured (like those of the 16F684 – see [mid-range lesson 10](#)), via the OSCCON register, to provide a range of processor clock frequencies:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OSCCON	–	IRCF3	IRCF2	IRCF1	IRCF0	–	SCS1	SCS0

The IRCF bits are used to select the internal oscillator frequency, as follows:

IRCF<3:0>	Oscillator	Frequency
000x	LF	31 kHz (approx)
001x	HF	31.25 kHz
0100	HF	62.5 kHz
0101	HF	125 kHz
0110	HF	250 kHz
0111	HF	500 kHz (default)
1011	HF	1 MHz
1100	HF	2 MHz
1101	HF	4 MHz
1110	HF	8 MHz
1111	HF	16 MHz

The 12F1501 actually has two internal RC oscillators: an uncalibrated low frequency oscillator, 'LFINTOSC', running at approximately 31 kHz, and a high frequency oscillator, 'HFINTOSC', which is factory-calibrated to run at 16 MHz.

This 16 MHz oscillator (twice the frequency of the 16F684's high frequency internal oscillator) is used as the clock source in the remaining "HF" modes, divided by a postscaler to generate frequencies down as low as 31.25 kHz, as shown in the table on the left⁷.

Unlike the mid-range PICs we've looked at, the default frequency is 500 kHz, instead of the usual 4 MHz.

The internal clock source (LFINTOSC or HFINTOSC, as above) is selected whenever the SCS1 bit is set, regardless of the processor configuration words.

Otherwise, if SCS<1:0> = 00, the clock source is selected by the oscillator selection bits in the configuration words.

Since the default processor clock frequency is now 500 kHz, if we want to use the previous delay loop code we would have to load the IRCF bits with '1101' to select a 4 MHz processor clock.

But why choose 4 MHz? Just because that's what we had to use on the 12F629? It makes more sense to select an oscillator frequency which will make it easier to implement our delay code.

The slower the processor runs, the fewer instruction cycles are needed to generate a given delay, which implies that slowing the clock will simplify the delay code.

Unfortunately, even with a 31 kHz clock, we'd still need two delay loops (one nested within the other), unless the loop is padded out with a lot of 'nop' or 'goto \$+1' instructions.

However, if we select a 1 MHz processor clock (for an instruction cycle time of 4 µs), the following code will generate a delay of approximately 500 ms:

```

; delay 500 ms
movlw .163 ; outer loop: 163 x (767 + 3) + 3
movwf dc2 ; = 125,513 cycles = 502.1 ms @ 4 us/cycle
clrf dc1 ; inner loop: 256 x 3 - 1
dly1 decfsz dc1,f ; = 767 cycles
goto dly1
decfsz dc2,f
goto dly1

```

That's only 7 instructions, compared with 11 in the 12F629 version – 36% shorter.

⁷ Not all possible IRCF values are shown here; those omitted duplicate some of the available processor frequencies.

Of course, we now have to select the oscillator frequency, as part of the initialisation routine:

```

; configure oscillator
movlw    b'01011010'      ; configure internal oscillator:
; -1011---                1 MHz (IRCF = 1011)
; -----1-              select internal clock (SCS = 1x)
banksel  OSCCON           ; -> 4 us / instruction cycle
movwf    OSCCON

```

But we would have had to do that anyway, if we'd wanted to select the “traditional” 4 MHz processor clock. And even if you choose to use the default 500 kHz clock, it's good practice to explicitly initialise the oscillator in any program, such as this one, which assumes a specific processor frequency – your code will be more likely to work (or at least you'll see more easily what has to be changed) if you later move it to another processor.

We also need to reserve data memory for the loop counter variables, `dc1` and `dc2`.

We had to use shared data memory in the 12F629 version, because all data memory on the 12F629 is unbanked. But on the 12F1501 we have a choice: these variables could be placed in either the (banked) general purpose RAM or the (unbanked) common RAM.

We only need one byte for each of these two variables, so there would be no problem with placing them in common RAM – and we wouldn't need to use a `banksel` directive before accessing them. But with only 16 bytes of common RAM available, even in the largest enhanced mid-range PIC devices, it's a scarce and therefore valuable resource. It's good to get into the habit of using common RAM only when you have a compelling reason to.

So we'll use a `UDATA` directive to declare our variables in the more-plentiful banked data memory:

```

;***** VARIABLE DEFINITIONS
UDATA
dc1      res 1              ; delay loop counters
dc2      res 1

```

And then use `banksel` before accessing these variables.

Finally, the 12F629 version used a shadow port register to avoid potential read-modify-write problems, but, as we've seen, we won't have any such problem with enhanced mid-range PICs if we modify output pins by operating on the associated LAT registers:

```

; toggle LED
banksel  LATA
movlw    1<<RA1           ; toggle LATA bit corresponding to RA1
xorwf    LATA,f

```

Complete program

The rest of the program is pretty much the same as the 12F629 version from [mid-range lesson 1](#); here's how it fits together:

```

;*****
; Description:      Migration lesson 1, example 2          *
;                                                          *
; Flashes an LED at approx 1 Hz.                        *
; LED continues to flash until power is removed.        *
;                                                          *
; Uses inline 500 ms delay routine                       *
;                                                          *
;*****

```



```

;*****
;
;   Pin assignments:
;       RA1 = indicator LED
;
;*****

#include "p12F1501.inc"

    errorlevel  -302          ; no warnings about registers not in bank 0
    errorlevel  -303          ; no warnings about program word too large

;***** CONFIGURATION
        ; ext reset, internal oscillator (no clock out), no watchdog,
        ;   brownout resets on, no power-up timer, no code protect
        ; no write protection, stack resets on, low brownout voltage,
        ;   no low-power brownout reset, high-voltage programming
    __CONFIG __CONFIG1, _MCLRE_ON & _FOSC_INTOSC & _CLKOUTEN_OFF & _WDTE_OFF &
    _BOREN_ON & _PWRTE_OFF & _CP_OFF
    __CONFIG __CONFIG2, _WRT_OFF & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF

;***** VARIABLE DEFINITIONS
        UDATA
dc1     res 1                ; delay loop counters
dc2     res 1

;***** RESET VECTOR *****
RES_VECT CODE    0x0000      ; processor reset vector

;***** MAIN PROGRAM *****

;***** Initialisation
start
        ; configure port
        movlw   ~(1<<RA1)    ; configure RA1 (only) as an output
        banksel TRISA
        movwf   TRISA

        ; configure oscillator
        movlw   b'01011010'  ; configure internal oscillator:
        ; -1011---          1 MHz (IRCF = 1011)
        ; -----1-        select internal clock (SCS = 1x)
        banksel OSCCON      ; -> 4 us / instruction cycle
        movwf   OSCCON

;***** Main loop
main_loop
        ; toggle LED
        banksel LATA
        movlw   1<<RA1        ; toggle LATA bit corresponding to RA1
        xorwf   LATA,f

        ; delay 500 ms
        banksel dc1          ; outer loop: 163 x (767 + 3) + 3
        movlw   .163         ; = 125,513 cycles
        movwf   dc2          ; = 502.1 ms @ 4 us/cycle

```

```

        clr     dc1          ; inner loop: 256 x 3 - 1
dly1   decfsz  dc1,f        ;   = 767 cycles
        goto   dly1
        decfsz  dc2,f
        goto   dly1

        ; repeat forever
        goto   main_loop

END

```

Subroutines and Modules

Subroutines, whether part of the main source code or called as an external module, operate the same way in the enhanced mid-range architecture as was described in [mid-range lesson 2](#).

In particular, paged access to program memory works the same way as in mid-range PICs, with the lower 11 bits of the program memory address being specified in the `goto` or `call` instruction opcode (giving a page size of 2048 words), while the upper bits are copied from the `PCLATH` register. The only difference is that, in the enhanced mid-range architecture, four bits of `PCLATH` (bits 3 to 6) are used to extend the address to 15 bits, making it possible to access up to 32k words of program memory, compared with 13-bit (8k word) addresses in mid-range devices.

Thus, although program memory can now be up to four times larger, the paging mechanism is essentially the same – and the `pagesel` directive should still be used to specify the page to jump to, as usual.

In [mid-range lesson 2](#), we developed a ‘`delay10`’ subroutine, which, assuming a 4 MHz processor clock, generates a delay of approximately $W \times 10$ ms. However, with the default processor frequency (derived from an internal RC oscillator) on enhanced mid-range devices being 500 kHz, it makes sense to re-implement this subroutine for an assumed 500 kHz clock.

For example:

```

;***** Variable delay: 10 ms to 2.55 s
;
; Delay = W x 10 ms
;
delay10                                ; delay = 2+Wx(223+1023+4)-1+4
        banksel dc1                    ;   = W x 1250 + 5 cycles
        movwf  dc2                      ;   = W x 10.0 ms @ 8 us/cycle
dly3   movlw  .74                        ; inner loop 1: 2 + 74 x 3 - 1
        movwf  dc1                      ;   = 223 cycles
dly1   decfsz  dc1,f
        goto   dly1
dly2   nop                                ; inner loop: 256 x 4 - 1
        decfsz  dc1,f                    ;   = 1023 cycles
        goto   dly2
        nop
        decfsz  dc2,f                    ; end outer loop
        goto   dly3

return

```

Otherwise, the examples from [mid-range lesson 2](#) need only a few changes, of types we’ve already seen (such as processor configuration, oscillator initialisation, use of LAT registers instead of shadow

variables) to work with the 12F1501 – so there would be little point in repeating them here. They are however available, as examples 3 and 4, in the source code provided with this lesson.

Digital Inputs

As mentioned earlier, the port registers (such as PORTA) operate the same way in the enhanced mid-range architecture as they do in mid-range devices, as described in [mid-range lesson 3](#).

If a pin is configured as a digital input, reading the corresponding bit in that pin's port register tells us whether the voltage present on that pin is 'high' or 'low'. To read the state of a digital input pin, read the corresponding port bit, as always.

The only difference is that, as we've seen, to modify an output pin you should update the corresponding data latch register. But reading the latch won't tell you anything about input pins; it will only tell you what the output pins are trying to output.

So, once again, we just need to remember, on the PIC12F1501:

- to read the current state of pins configured as a digital inputs, read PORTA
- to initialise or modify pins configured as digital outputs, write to or update LATA

Other than the use of LATA, the examples from [mid-range lesson 3](#) do not need many changes to work with the 12F1501 – and of course there is no need to repeat topics such as switch debouncing techniques, which are the same across architectures. So we'll only look briefly at a couple of examples of using digital inputs, to illustrate the few differences that apply to enhanced mid-range PICs.

Example 5: Reading a Pushbutton Switch

The first example in [mid-range lesson 3](#) simply turned on the LED on RA1 on whenever the pushbutton on RA3 was pressed.

The solution is fairly simple: test the bit in PORTA corresponding to RA3 to read the switch, and if the RA3 bit is clear (indicating a button press), set the RA1 bit in the LATA register. Otherwise, the button is not pressed, so clear RA1 in LATA.

Here is the main loop, implementing this logic:

```
main_loop
    ; turn on LED only if button pressed
    banksel PORTA
    btfsc  PORTA,RA3      ; if button pressed (RA3 low)
    goto  btn_up
    banksel LATA
    bsf    LATA,RA1      ; turn on LED
    goto  btn_end
btn_up
    banksel LATA          ; else
    bcf    LATA,RA1      ; turn off LED
btn_end

    ; repeat forever
    goto  main_loop
```

The important point here is that we are reading (testing) PORTA, but only writing to LATA – avoiding the need to use a shadow port register.

Although it's not really necessary in this example to initialise the output pin state before entering the main loop, it's good practice to ensure that outputs will be in a desired initial state when they are first enabled.

So our initialisation code becomes:

```

; configure port
banksel LATA          ; start with all output pins low
clrf   LATA
movlw  ~(1<<RA1)     ; configure RA1 (only) as an output
banksel TRISA        ; (RA3 is an input)
movwf  TRISA

```

Note that it would be equally valid to write to **PORTA** here; we're clearing the entire port, not modifying individual bits, so read-modify-write is not a consideration here. But by sticking to the rule "always write to the latch", it makes it easy to always get it right – there is no need to think about whether read-modify-write considerations might apply or not.

Complete program

Here is the complete 12F1501 version of the turning on the LED when the pushbutton is pressed program:

```

;*****
;
; Description:      Migration lesson 1, example 5
;
; Demonstrates reading a switch
;
; Turns on LED when pushbutton is pressed
;
;*****
;
; Pin assignments:
; RA1 = LED
; RA3 = pushbutton switch (active low)
;
;*****
#include "p12F1501.inc"

errorlevel -302      ; no warnings about registers not in bank 0
errorlevel -303      ; no warnings about program word too large

;***** CONFIGURATION
; int reset, internal oscillator (no clock out), no watchdog,
; brownout resets on, no power-up timer, no code protect
; no write protection, stack resets on, low brownout voltage,
; no low-power brownout reset, high-voltage programming
__CONFIG __CONFIG1, _MCLRE_OFF & _FOSC_INTOSC & _CLKOUTEN_OFF & _WDTE_OFF &
_BOREN_ON & _PWRTE_OFF & _CP_OFF
__CONFIG __CONFIG2, _WRT_OFF & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF

;***** RESET VECTOR *****
RES_VECT CODE 0x0000 ; processor reset vector

;***** MAIN PROGRAM *****

```

```

;***** Initialisation
start
    ; configure port
    banksel LATA                ; start with all output pins low
    clrf    LATA
    movlw   ~(1<<RA1)          ; configure RA1 (only) as an output
    banksel TRISA                ; (RA3 is an input)
    movwf   TRISA

;***** Main loop
main_loop
    ; turn on LED only if button pressed
    banksel PORTA
    btfsc   PORTA,RA3          ; if button pressed (RA3 low)
    goto    btn_up
    banksel LATA
    bsf     LATA,RA1           ; turn on LED
    goto    btn_end

btn_up
    banksel LATA                ; else
    bcf     LATA,RA1           ; turn off LED

btn_end

    ; repeat forever
    goto    main_loop

END

```

Example 6: Internal Pull-ups

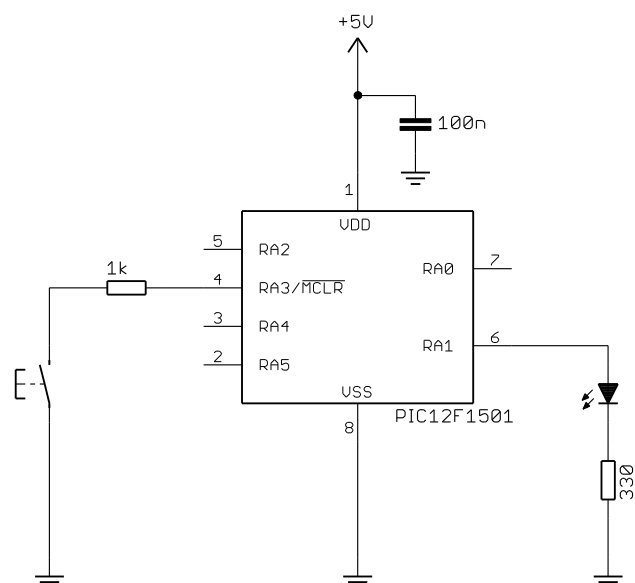
The PIC12F1501 makes weak pull-ups available for use with digital inputs, in much the same way as we saw in [mid-range lesson 3](#) for the 12F629.

In the pull-up example in that lesson, we toggled an LED each time a pushbutton switch was pressed.

The 12F1501 provides individually-selectable pull-ups on every pin.

This means that, unlike the example in [mid-range lesson 3](#), we can continue to use RA3 as our pushbutton input, as shown in the circuit on the right.

If you are using the [Gooligum training board](#), simply remove jumper JP3 to disconnect the external pull-up resistor from the pushbutton on RA3.



As in the mid-range architecture, the internal weak pull-ups are controlled as a group by a global enable bit, now known as $\overline{\text{WPUEN}}$, in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION_REG	$\overline{\text{WPUEN}}$	INTEDG	TMR0CS	TMR0SE	PSA	PS2	PS1	PS0

By default (after a power-on or reset), $\overline{\text{WPUEN}} = 1$ and all the internal pull-ups are disabled.

To globally enable internal pull-ups, clear $\overline{\text{WPUEN}}$.

Each weak pull-up is then individually controlled by a bit in the WPUA register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
WPUA	-	-	WPUA5	WPUA4	WPUA3	WPUA2	WPUA1	WPUA0

If $\text{WPUA}\langle n \rangle = 1$, the weak pull-up on the corresponding PORTA pin, RAn , is enabled.

If $\text{WPUA}\langle n \rangle = 0$, the corresponding weak pull-up is disabled.

However, if a pin is configured as an output, the internal pull-up is automatically disabled for that pin.

To enable the pull-up on RA3, we must first clear $\overline{\text{WPUEN}}$ to globally enable weak pull-ups:

```
banksel OPTION_REG          ; enable global pull-ups
bcf     OPTION_REG, NOT_WPUEN
```

Then, having globally enabled weak pull-ups, we need to enable the individual pull-up on RA3, by setting $\text{WPUA}\langle 3 \rangle$.

You could do that by:

```
bsf     WPUA, WPUA3          ; enable pull-up on RA3
```

As in mid-range PICs, every bit of WPUA is set by default, so there is no real need to explicitly set $\text{WPUA}\langle 3 \rangle$ like this. But it's good practice to disable the weak pull-ups on the unused input pins. Therefore, all the remaining bits in WPUA should be cleared.

This could be done by:

```
clrf    WPUA                 ; disable all pull-ups
bsf     WPUA, WPUA3          ; except on RA3
```

or:

```
movlw   1<<WPUA3             ; enable pull-up on RA3 only
movwf   WPUA
```

The second form is better if you need to enable pull-ups on more than one input.

Alternatively, you can express this as:

```
movlw   1<<RA3               ; enable pull-up on RA3 only
movwf   WPUA
```

because the 'p12F1501.inc' processor include file defines both 'WPUA3' and 'RA3' as constants representing the value '3'. Using 'RA3' can make it clearer that you're configuring pin RA3.

The pull-up example in [mid-range lesson 3](#) used a counting algorithm to debounce the switch.

We could re-use the code from that example, but because it was written for a PIC12F629 running at its default 4 MHz processor clock, we'd have to configure our PIC12F1501 to also run at 4 MHz. That's a reasonable approach, but it would be better to continue to use the enhanced mid-range architecture's default 500 kHz processor clock – which means changing the delays in the debouncing code, in line with the lower clock rate, as follows:

```

        ; wait for button release, debounce by counting:
db_up   movlw   .10                ; max count = 10ms/992us = 10
        banksel db_cnt
        movwf  db_cnt
up_dly  movlw   .41                ; delay 41x3+1 = 124 cycles = 992 us.
        movwf  dcl
up_dly2 decfsz  dcl,f
        goto   up_dly2
        banksel PORTA
        btfss  PORTA,RA3          ; if button down (RA3 low),
        goto   db_up              ; restart count
        banksel db_cnt
        decfsz db_cnt,f           ; else repeat until max count reached
        goto   up_dly

```

Complete program

Here's the complete "Toggle an LED" program, illustrating how to read and debounce a simple switch on a digital input pin held high by an internal pull-up:

```

;*****
;
; Description:      Migration lesson 1, example 6
;
; Demonstrates use of internal pullups plus debouncing
;
; Toggles LED when pushbutton is pressed then released,
; using a counting algorithm to debounce switch
;
;*****
;
; Pin assignments:
; RA1 = LED
; RA3 = pushbutton switch (active low)
;
;*****
#include "p12F1501.inc"

errorlevel -302          ; no warnings about registers not in bank 0
errorlevel -303          ; no warnings about program word too large

;***** CONFIGURATION
        ; int reset, internal oscillator (no clock out), no watchdog,
        ; brownout resets on, no power-up timer, no code protect
        ; no write protection, stack resets on, low brownout voltage,
        ; no low-power brownout reset, high-voltage programming
__CONFIG __CONFIG1, _MCLRE_OFF & _FOSC_INTOSC & _CLKOUTEN_OFF & _WDTE_OFF &
__BOREN_ON & _PWRTE_OFF & _CP_OFF
__CONFIG __CONFIG2, _WRT_OFF & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF

```

```

;***** VARIABLE DEFINITIONS
        UDATA
db_cnt  res 1          ; debounce counter
dcl     res 1          ; delay counter

;***** RESET VECTOR *****
RES_VECT CODE    0x0000      ; processor reset vector

;***** MAIN PROGRAM *****

;***** Initialisation
start
        ; configure port
        banksel LATA          ; start with LED off
        clrf    LATA
        movlw   ~(1<<RA1)    ; configure RA1 (only) as an output
        banksel TRISA        ; (RA3 is an input)
        movwf  TRISA
        banksel OPTION_REG   ; enable global pull-ups
        bcf    OPTION_REG,NOT_WPUEN
        movlw  1<<RA3        ; enable pull-up on RA3 only
        banksel WPUA
        movwf  WPUA

        ; configure oscillator
        movlw  b'00111010'   ; configure internal oscillator:
        ; -0111---          500 kHz (IRCF = 0111)
        ; -----1-        select internal clock (SCS = 1x)
        banksel OSCCON      ; -> 8 us / instruction cycle
        movwf  OSCCON

;***** Main loop
main_loop
        ; wait for button press
        banksel PORTA
wait_dn  btfsc  PORTA,RA3    ; wait until RA3 low
        goto   wait_dn

        ; toggle LED
        banksel LATA
        movlw  1<<RA1        ; toggle LATA bit corresponding to RA1
        xorwf  LATA,f

        ; wait for button release, debounce by counting:
db_up    movlw  .10          ; max count = 10ms/992us = 10
        banksel db_cnt
        movwf  db_cnt
up_dly   movlw  .41          ; delay 41x3+1 = 124 cycles = 992 us.
        movwf  dcl
up_dly2  decfsz dcl,f
        goto   up_dly2
        banksel PORTA
        btfss  PORTA,RA3    ; if button down (RA3 low),
        goto   db_up        ; restart count
        banksel db_cnt
        decfsz db_cnt,f     ; else repeat until max count reached
        goto   up_dly

```



```
; repeat forever  
goto    main_loop
```

```
END
```

Conclusion

This lesson has shown that enhanced mid-range PICs are essentially similar to the mid-range devices we're familiar with, but are certainly "enhanced" in a number of useful ways.

In particular, we've seen that the enhanced mid-range architecture:

- supports a larger data memory address space, via a new bank selection register
- supports a larger program memory address space, through an extended program counter
- provides data latch registers which avoid potential read-modify-write issues when modifying output pins
- has a default processor clock frequency, when using the internal RC oscillator, of 500 kHz instead of 4 MHz (although on the 12F1501 this is configurable)

Otherwise the instruction set is much the same as before (with a few new instructions added) and facilities such as internal weak pull-ups are largely unchanged, but may be more flexible than before (weak pull-ups being available on every pin on the 12F1501, compared with not being available on GP3 on the 12F629).

Another facility that is largely unchanged in enhanced mid-range devices is the 8-bit Timer0 module, which we will take a brief look at in the next lesson, before introducing the enhanced mid-range architecture's interrupt-handling capability – in particular its ability to automatically save and restore the processor context.

Migrating to the Enhanced Mid-Range PIC Architecture

Programming Enhanced Mid-Range PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 1: Basic Digital I/O

This series of lessons introduces the enhanced mid-range PIC architecture, using C, and follows on from the [mid-range PIC C](#) tutorial series. It assumes that you are familiar with the content of at least the free mid-range C lessons (1-5).

Although the mid-range architecture was a step up from the baseline architecture, it still has a number of limitations that we've had to work around, such as the read-modify-write problem and the need to address data memory in contiguous blocks of no more than 96 bytes.

The enhanced mid-range (12F1xxx and 16F1xxx) PIC architecture overcomes these limitations and more, removing the need to use shadow registers, extending the maximum code and data memory sizes, and including architectural enhancements which make the code generated by the compiler more efficient¹.

This lesson introduces one of the simplest of the enhanced mid-range PICs – the PIC12F1501. It then goes on to describe basic digital I/O output, as was covered in [mid-range C lesson 1](#).

In summary, this lesson covers:

- Introduction to the PIC12F1501
- Simple digital input and output
- Selecting the internal oscillator frequency
- Using internal (weak) pull-ups

Getting Started

As before, these tutorials assume that you are using either the [Gooligum Baseline and Mid-range PIC Training and Development Board](#) or Microchip's Low Pin Count Demo Board², with Microchip's MPLAB 8 or MPLAB X integrated development environment.

You will also need a programmer, such as Microchip's PICkit 3, which is compatible the enhanced mid-range PICs.

*Note: the PICkit 2 programmer, which was used in the baseline and mid-range tutorials, is **not supported** in MPLAB for use with enhanced mid-range PICs.*

¹ if you're interested in the detail of the enhanced mid-range PIC architecture and how it compares with "ordinary" mid-range PICs, see [enhanced mid-range assembler migration lesson 1](#)

² it is of course possible to adapt these lessons to different development boards

Introducing the PIC12F1501

The 12F1501 is in some ways the simplest of the enhanced mid-range PICs³.

It is roughly equivalent to the 12F675, which in turn is essentially the same as the 12F629 introduced in [mid-range assembler lesson 1](#), with the addition of an analog-to-digital converter (ADC) – as can be seen in the following table, which summarises some of the basic features of various 8-pin PICs:

Device	Memory (words or bytes)			Timers		Analog		Clock rate (max MHz)
	Program	Data	EEPROM	8-bit	16-bit	Comp-arators	ADC inputs	
12F508	512	25	0	1	0	0	0	4
12F629	1024	64	128	1	1	1	0	20
12F675	1024	64	128	1	1	1	4	20
12F683	2048	128	256	2	1	1	4	20
12F1501	1024	64	0	2	1	1	4	20
12F1822	2048	128	256	2	1	1	4	32
12F1840	4096	256	256	2	1	1	4	32

However, the 12F1501 also includes a number of peripherals which the 12F675 does not, such as a digital-to-analog converter (DAC), pulse-width modulation (PWM) outputs, a configurable logic cell (used to implement simple logic functions in hardware) and a numerically controlled oscillator.

And although the 12F1501 does not include any EEPROM memory, it has the ability to write into its program (Flash) memory, 128 bytes of which is “high-endurance” Flash which can be re-written at least 100,000 times – making it a viable alternative to true EEPROM memory in many situations.

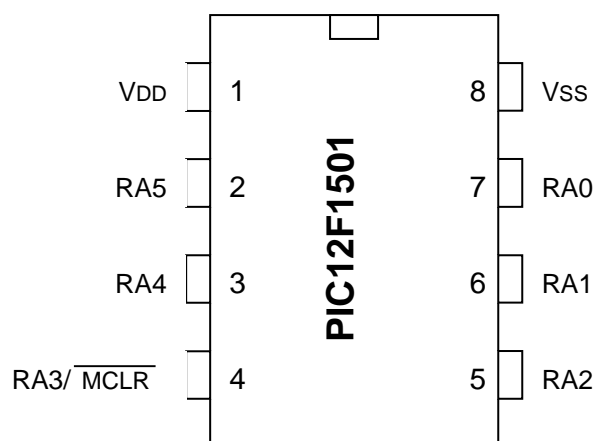
We’ll explore these additional features in later lessons.

Like all 12F PICs, the 12F1501 provides six I/O pins in an eight-pin package.

VDD is the positive power supply and VSS is the “negative” supply, or ground.

VDD (relative to VSS) on the PIC12F1501 can range from 2.3 V to 5.5 V, although at least 2.5 V is needed if the clock rate is greater than 16 MHz⁴.

The remaining pins, RA0 to RA5, are the I/O pins. Just like the baseline and mid-range 8-pin PICs, they are used for digital input and output, except for RA3, which can only be used as an input. The other pins can be individually set to be inputs or outputs.



³ which is why we’re starting this series with the PIC12F1501...

⁴ A low-power variant, the PIC12LF1501, is also available, where VDD can range from 1.8 V to 3.6 V, with at least 2.5 V needed for clock rates above 16 MHz.

Each of these I/O pins has more functions that can be assigned to it – too many to show on the diagram above. As we’ll see in a later lesson, some of these functions can be selectively mapped to alternate pins. And as usual, some functions may need to be disabled before a pin can be used for digital I/O.

Taken together, the six I/O pins comprise the general-purpose I/O *port*, or GPIO port, which is referred to as **PORTA** on the 12F1501.

As in the midrange architecture, there is a TRIS register (**TRISA**) associated with the port which controls whether each pin is set as an input or output:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TRISA			TRISA5	TRISA4		TRISA2	TRISA1	TRISA0

It works the same way as the TRIS registers we’ve seen before, with a ‘1’ configuring the corresponding pin as an input, and a ‘0’ configuring it as an output. And as usual, every pin is configured as an input by default; **TRISA** is set to all ‘1’s when the device is powered on.

As you’d expect, there is also a “**PORTA**” register which provides access to the port pins:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTA			RA5	RA4	RA3	RA2	RA1	RA0

If a pin is configured as an output, setting the corresponding **PORTA** bit to ‘1’ outputs a ‘high’ on that pin; clearing it to ‘0’ outputs a ‘low’.

Reading the **PORTA** register reads the voltage present on each pin, assuming that the pins being read are configured for digital I/O⁵. If the voltage on a pin is high, the corresponding bit reads as ‘1’; if the input pin is low, the corresponding bit reads as ‘0’.

Note: the port registers represent the actual voltages present on each digital I/O pin, including pins configured as digital outputs

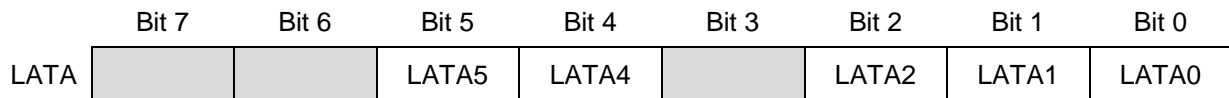
This behaviour is the same as in the baseline and mid-range PIC architectures. If you attempt to output a ‘high’ on a pin by writing a ‘1’ to the corresponding port bit, but the external circuit holds that pin low, that pin will read as ‘0’ – not what you might have expected.

As was explained in [baseline lesson 2](#), this behaviour can lead to what are known as *read-modify-write* problems, where instructions which are intended to modify only specific pins actually read the entire port, including pins which may not reflect the value that had been output to them, and then write the new value (with some bits possibly incorrect) back to the port.

To work around this potential problem, we have been using *shadow registers*, to keep track of what we have output to the port pins, and to operate on that copy, periodically writing it out to the port.

⁵ that is, any functionality on a pin which interferes with digital I/O operation, such as being configured as an analog input (see for example [baseline lesson 10](#)), has been disabled

To avoid read-modify-write problems, the enhanced mid-range architecture makes available an “output data latch” register, associated with each port:



Writing to LATA has the same effect as writing to PORTA: if a pin is configured as an output, setting the corresponding LATA bit to ‘1’ outputs a ‘high’ on that pin; clearing it to ‘0’ outputs a ‘low’.

However – reading LATA returns the value that was last written to LATA. It does not read the voltages on the pins (whether input or output) themselves.

This means that **in the enhanced mid-range architecture, there is no need to use shadow registers**, as long as you follow these rules:

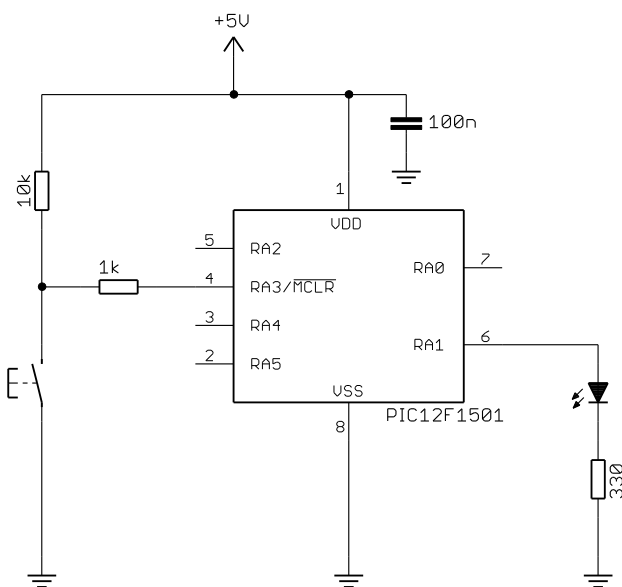
- if you are writing an entire byte to a port, you can write to either PORTA or LATA
- if you are modifying individual port pins, you should operate on LATA
- if you are reading digital input pins, you must read PORTA

To keep it simpler, you won’t run into any problems if you always access LATA to write to or modify output pins, and PORTA to read digital input pins.

Example 1: Turning on an LED

We’ll start the same way that we did in [mid-range C lesson 1](#), by simply lighting a single LED, connected to one of the 12F1501’s digital I/O pins.

And we’ll use the same circuit as before:



In the same way as the 8-pin baseline and mid-range PICs, pin 4 can be configured as either a digital input (RA3) or as an external reset (“master clear”, $\overline{\text{MCLR}}$), which, if pulled low, will reset the processor.

In this example, we’ll configure the PIC for external reset, allowing either the pushbutton or PICkit 3 to pull $\overline{\text{MCLR}}$ low, resetting the device.

If you are using the [Gooligum training board](#), plug the PIC12F1501 into the top section of the 14-pin IC socket – the section marked ‘12F’. Close jumpers JP3 and JP12 to bring the 10 k Ω resistor into the circuit and to connect the LED to RA1, and ensure that every other jumper is disconnected.

With this simple circuit in place, and connected to your PC via a suitable programmer, such as a PICkit 3, it’s time to move on to programming!

As usual, after the comment block at the start of the program, we begin with:

```
#include <xc.h>
```

Next the processor is configured:

```
/****** CONFIGURATION *****/  
// ext reset, internal oscillator (no clock out), no watchdog timer  
#pragma config MCLRE = ON, FOSC = INTOSC, CLKOUTEN = OFF, WDTE = OFF  
// brownout resets enabled, low brownout voltage, no low-power brownout reset  
#pragma config BOREN = ON, BORV = LO, LPBOR = OFF  
// no power-up timer, no code protect, no write protection  
#pragma config PWRTE = OFF, CP = OFF, WRT = OFF  
// stack resets on, high-voltage programming  
#pragma config STVREN = ON, LVP = OFF
```

We've seen many of these configuration options before, and those we haven't we'll examine in greater detail in later lessons, but briefly the options being selected here are:

- MCLRE = ON
Enables the external reset, or “master clear” ($\overline{\text{MCLR}}$) on pin 4.
- FOSC = INTOSC
Selects the internal RC oscillator as the clock source
- CLKOUTEN = OFF
Disables the clock out function on pin 3
- WDTE = OFF
Disables the watchdog timer.
- BOREN = ON
Enables brown-out reset.
- BORV = LO
Selects the low brown-out reset voltage option
- LPBOR = OFF
Disables the low-power brown-out reset facility.
- PWRTE = OFF
Disables the power-up timer.
- CP = OFF
Turns off program memory code protection.
- WRT = OFF
Disables flash memory write protection.
- STVREN = ON
Enables stack overflow/underflow resets.

The stack is managed by the C compiler and should not be something that you usually have to be concerned about. If a stack overflow or underflow does occur, it means that something has gone wrong and your program probably isn't working properly. If this option is enabled, the PIC will reset itself if a stack overflow or underflow occurs – hopefully allowing your program to recover.

- LVP = OFF
Disables low-voltage programming.

To program the device, a high voltage (around 12 V) must be applied to the VPP pin, as was the case with the baseline and mid-range PICs we've examined. Low-voltage programming mode avoids the need for this high voltage, but we don't need it because the PICkit 3 can operate in the “normal” high-voltage programming mode.

Within the main program, we start by configuring the RA1 pin as an output, as we've done many times before:

```
TRISA = ~(1<<1);          // configure RA1 (only) as an output
```

To make RA1 output a 'high', we have to set bit 1 of PORTA to '1', which we could do with:

```
PORTA = 0b000010;       // set RA1 high
```

because there is no risk of running into read-modify-write problems when writing an entire byte, updating the port register in a single operation, like this.

However, it is better to get into the habit of writing to LATA to modify output pins.

In particular, by using LATA, we can safely modify individual bits without having to be concerned about whether it might be a read-modify-write operation.

So on an enhanced mid-range device, we can safely use:

```
LATAbits.LATA1 = 1;      // set RA1 high
```

Finally, we need an "infinite loop", to stop the program running off into the rest of (uninitialised) program memory:

```
for (;;)
{
    // loop forever
    ;
}
```

Complete program

Putting together all the above, here's our complete assembler source for turning on an LED:

```

/*****
*
*   Description:      Migration lesson 1, example 1
*
*   Turns on LED.   LED remains on until power is removed.
*
*****/
*
*   Pin assignments:
*       RA1 = indicator LED
*
*****/

#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, internal oscillator (no clock out), no watchdog timer
#pragma config MCLRE = ON, FOSC = INTOSC, CLKOUTEN = OFF, WDTE = OFF
// brownout resets enabled, low brownout voltage, no low-power brownout reset
#pragma config BOREN = ON, BORV = LO, LPBOR = OFF
// no power-up timer, no code protect, no write protection
#pragma config PWRTE = OFF, CP = OFF, WRT = OFF
// stack resets on, high-voltage programming
#pragma config STVREN = ON, LVP = OFF

```

```

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    TRISA = ~(1<<1);    // configure RA1 (only) as an output

    LATAbits.LATA1 = 1;    // set RA1 high

    /*** Main loop
    for (;;)
    {
        // loop forever
        ;
    }
}

```

Example 2: Flashing an LED (50% duty cycle)

Having lit a single LED, the next step is to make it flash (as always...).

Although it usually makes more sense to use a timer (see for example [mid-range C lesson 2](#), or, using interrupts, [mid-range C lesson 3](#)) to time a process such as flashing an LED, you might occasionally want to use in-line delays, as we did in [mid-range C lesson 1](#), where we toggled an LED every 500 ms.

Recall that the XC8 compiler provides two macros: ‘`__delay_us()`’ and ‘`__delay_ms()`’, which use the ‘`__delay(n)`’ function create delays specified in μs and ms respectively and that to do so, they reference the symbol “`_XTAL_FREQ`”, which must be defined as the processor oscillator frequency, in Hertz.

The PIC12F629 used in the early mid-range C lessons has an internal RC oscillator which can only provide a fixed 4 MHz processor.

However, the 12F1501’s internal RC oscillators can be configured (like those of the 16F684 – see mid-range lesson 10), via the `OSCCON` register, to provide a range of processor clock frequencies:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OSCCON	–	IRCF3	IRCF2	IRCF1	IRCF0	–	SCS1	SCS0

The IRCF bits are used to select the internal oscillator frequency, as follows:

IRCF<3:0>	Oscillator	Frequency
000x	LF	31 kHz (approx)
001x	HF	31.25 kHz
0100	HF	62.5 kHz
0101	HF	125 kHz
0110	HF	250 kHz
0111	HF	500 kHz (default)
1011	HF	1 MHz
1100	HF	2 MHz
1101	HF	4 MHz
1110	HF	8 MHz
1111	HF	16 MHz

The 12F1501 actually has two internal RC oscillators: an uncalibrated low frequency oscillator, ‘LFINTOSC’, running at approximately 31 kHz, and a high frequency oscillator, ‘HFINTOSC’, which is factory-calibrated to run at 16 MHz.

This 16 MHz oscillator (twice the frequency of the 16F684’s high frequency internal oscillator) is used as the clock source in the remaining “HF” modes, divided by a postscaler to generate frequencies down as low as 31.25 kHz, as shown in the table on the left⁶.

Unlike the mid-range PICs we’ve looked at, the default frequency is 500 kHz, instead of the usual 4 MHz.

The internal clock source (LFINTOSC or HFINTOSC, as above) is selected whenever the SCS1 bit is set, regardless of the processor configuration words.

Otherwise, if SCS<1:0> = 00, the clock source is selected by the oscillator selection bits in the configuration words.

The processor clock frequency isn’t really important in this example – any of these (even 31 kHz) is fast enough to flash an LED.

But it’s important to be aware of what frequency the processor is running at, so that you can correctly define the “_XTAL_FREQ” symbol. And although we are using the default 500 kHz clock, it’s good practice to explicitly initialise the oscillator in any program, such as this one, which assumes a specific processor frequency – your code will be more likely to work (or at least you’ll see more easily what has to be changed) if you later move it to another processor.

So we should include in our initialisation routine:

```
// configure oscillator
OSCONbits.SCS1 = 1;           // select internal clock
OSCONbits.IRCF = 0b0111;     // internal oscillator = 500 kHz
```

Finally, the 12F629 version used a shadow port register to avoid potential read-modify-write problems, but, as we’ve seen, we won’t have any such problem with enhanced mid-range PICs if we modify output pins by operating on the associated LAT registers:

```
// toggle LED on RA1
LATAbits.LATA1 = ~LATAbits.LATA1;
```

⁶ Not all possible IRCF values are shown here; those omitted duplicate some of the available processor frequencies.

Complete program

The rest of the program is pretty much the same as the 12F629 version from [mid-range C lesson 1](#); here's how it fits together:

```

/*****
*
*   Description:      Migration lesson 1, example 2
*
*   Flashes an LED at approx 1 Hz.
*   LED continues to flash until power is removed.
*
*****/
*
*   Pin assignments:
*       RA1 = flashing LED
*
*****/

#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, internal oscillator (no clock out), no watchdog timer
#pragma config MCLRE = ON, FOSC = INTOSC, CLKOUTEN = OFF, WDTE = OFF
// brownout resets enabled, low brownout voltage, no low-power brownout reset
#pragma config BOREN = ON, BORV = LO, LPBOR = OFF
// no power-up timer, no code protect, no write protection
#pragma config PWRTE = OFF, CP = OFF, WRT = OFF
// stack resets on, high-voltage programming
#pragma config STVREN = ON, LVP = OFF

#define _XTAL_FREQ 500000 // oscillator frequency for _delay()

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    LATA = 0; // start with all output pins low (LED off)
    TRISA = ~(1<<1); // configure RA1 (only) as an output

    // configure oscillator
    OSCCONbits.SCS1 = 1; // select internal clock
    OSCCONbits.IRCF = 0b0111; // internal oscillator = 500 kHz

    /*** Main loop
    for (;;)
    {
        // toggle LED on RA1
        LATAbits.LATA1 = ~LATAbits.LATA1;

        // delay 500 ms
        __delay_ms(500);

    } // repeat forever
}

```

Digital Inputs

As mentioned earlier, the port registers (such as PORTA) operate the same way in the enhanced mid-range architecture as they do in mid-range devices, as described in [mid-range assembler lesson 3](#).

If a pin is configured as a digital input, reading the corresponding bit in that pin's port register tells us whether the voltage present on that pin is 'high' or 'low'. To read the state of a digital input pin, read the corresponding port bit, as always.

The only difference is that, as we've seen, to modify an output pin you should update the corresponding data latch register. But reading the latch won't tell you anything about input pins; it will only tell you what the output pins are trying to output.

So, once again, we just need to remember, on the PIC12F1501:

- to read the current state of pins configured as a digital inputs, read PORTA
- to initialise or modify pins configured as digital outputs, write to or update LATA

Other than the use of LATA, the digital input examples from [mid-range C lesson 1](#) do not need many changes to work with the 12F1501 – and of course there is no need to repeat topics such as switch debouncing techniques, which are the same across architectures. So we'll only look briefly at a couple of examples of using digital inputs, to illustrate the few differences that apply to enhanced mid-range PICs.

Example 3: Reading a Pushbutton Switch

The first digital input example in [mid-range C lesson 1](#) simply turned on the LED on RA1 on whenever the pushbutton on RA3 was pressed.

The solution is fairly simple: test the bit in PORTA corresponding to RA3 to read the switch, and if the RA3 bit is clear (indicating a button press), set the RA1 bit in the LATA register. Otherwise, the button is not pressed, so clear RA1 in LATA.

We can copy the state of the RA3 input (inverted because the switch is active low) to the RA1 output simply and directly with:

```
LATABits.LATA1 = ~PORTABits.RA3;    // copy ~RA3 input to RA1 output
```

The important point here is that we are reading (testing) PORTA, but only writing to LATA – avoiding the need to use a shadow port register.

Although it's not really necessary in this example to initialise the output pin state before entering the main loop, it's good practice to ensure that outputs will be in a desired initial state when they are first enabled.

So our initialisation code becomes:

```
// configure port
LATA = 0;                // start with all output pins low (LED off)
TRISA = ~(1<<1);        // configure RA1 (only) as an output
```

Note that it would be equally valid to write to PORTA here; we're clearing the entire port, not modifying individual bits, so read-modify-write is not a consideration here. But by sticking to the rule "always write to the latch", it makes it easy to always get it right – there is no need to think about whether read-modify-write considerations might apply or not.

Complete program

Here is the complete 12F1501 version of the turning on the LED when the pushbutton is pressed program:

```

/*****
*
*   Description:      Migration lesson 1, example 3
*
*   Demonstrates reading a switch
*
*   Turns on LED when pushbutton is pressed
*
*****/
*
*   Pin assignments:
*       RA1 = LED
*       RA3 = pushbutton switch (active low)
*
*****/

#include <xc.h>

/***** CONFIGURATION *****/
// int reset, internal oscillator (no clock out), no watchdog timer
#pragma config MCLRE = OFF, FOSC = INTOSC, CLKOUTEN = OFF, WDTE = OFF
// brownout resets enabled, low brownout voltage, no low-power brownout reset
#pragma config BOREN = ON, BORV = LO, LPBOR = OFF
// no power-up timer, no code protect, no write protection
#pragma config PWRTE = OFF, CP = OFF, WRT = OFF
// stack resets on, high-voltage programming
#pragma config STVREN = ON, LVP = OFF

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    LATA = 0;                // start with all output pins low (LED off)
    TRISA = ~(1<<1);        // configure RA1 (only) as an output

    /*** Main loop
    for (;;)
    {
        // turn on LED only if button pressed
        LATAbits.LATA1 = ~PORTAbits.RA3;
    }
}

```

Example 4: Internal Pull-ups

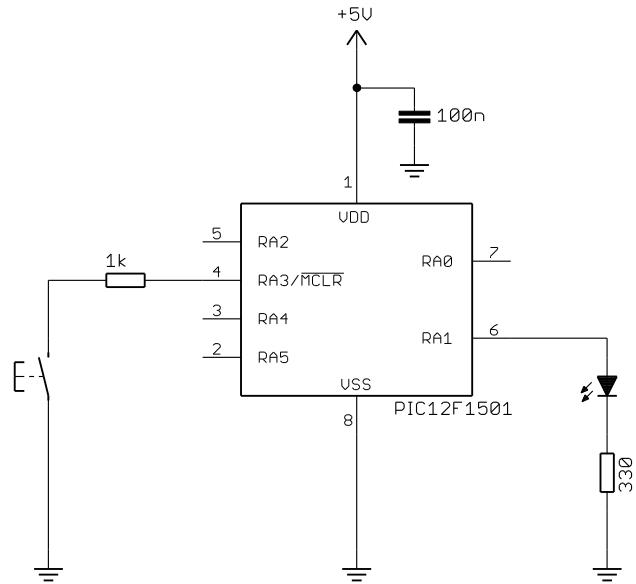
The PIC12F1501 makes weak pull-ups available for use with digital inputs, in much the same way as we saw in [mid-range C lesson 1](#) for the 12F629.

In the pull-up example in that lesson, we toggled an LED each time a pushbutton switch was pressed.

The 12F1501 provides individually-selectable pull-ups on every pin.

This means that, unlike the example in [mid-range C lesson 1](#), we can continue to use RA3 as our pushbutton input, as shown in the circuit on the right.

If you are using the [Gooligum training board](#), simply remove jumper JP3 to disconnect the external pull-up resistor from the pushbutton on RA3.



As in the mid-range architecture, the internal weak pull-ups are controlled as a group by a global enable bit, now known as $\overline{\text{WPUEN}}$, in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION_REG	$\overline{\text{WPUEN}}$	INTEDG	TMR0CS	TMR0SE	PSA	PS2	PS1	PS0

By default (after a power-on or reset), $\overline{\text{WPUEN}} = 1$ and all the internal pull-ups are disabled. To globally enable internal pull-ups, clear $\overline{\text{WPUEN}}$.

Each weak pull-up is then individually controlled by a bit in the WPUA register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
WPUA	-	-	WPUA5	WPUA4	WPUA3	WPUA2	WPUA1	WPUA0

If $\text{WPUA}\langle n \rangle = 1$, the weak pull-up on the corresponding PORTA pin, RAn , is enabled.

If $\text{WPUA}\langle n \rangle = 0$, the corresponding weak pull-up is disabled.

However, if a pin is configured as an output, the internal pull-up is automatically disabled for that pin.

To enable the pull-up on RA3, we must first clear $\overline{\text{WPUEN}}$ to globally enable weak pull-ups:

```
OPTION_REGbits.nWPUEN = 0; // enable weak pull-ups (global)
```

As always you should look at the header file for your PIC ("pic12f1501.h" in this case) to check the name of the bit-field ('nWPUEN') associated with the register bit ($\overline{\text{WPUEN}}$) you wish to access.

Then, having globally enabled weak pull-ups, we need to enable the individual pull-up on RA3, by setting $\text{WPUA}\langle 3 \rangle$.

You could do that by:

```
WPUAbits.WPUA3 = 1; // enable pull-up on RA3
```

As in mid-range PICs, every bit of WPUA is set by default, so there is no real need to explicitly set WPUA<3> like this. But it's good practice to disable the weak pull-ups on the unused input pins. Therefore, all the remaining bits in WPUA should be cleared.

This could be done by:

```
WPUA = 1<<3;           // enable pull-up on RA3 only
```

The pull-up example in [mid-range C lesson 1](#) used a counting algorithm to debounce the switch, which we can re-use here.

Complete program

Here's the complete "Toggle an LED" program, slightly modified from that in [mid-range C lesson 1](#) and including the changes shown above, illustrating how to read and debounce a simple switch on a digital input pin held high by an internal pull-up:

```

/*****
 *
 * Description:      Migration lesson 1, example 4
 *
 * Demonstrates use of internal pullups plus debouncing
 *
 * Toggles LED when pushbutton is pressed then released,
 * using a counting algorithm to debounce switch
 *
 *****/
 *
 * Pin assignments:
 *   RA1 = LED
 *   RA3 = pushbutton switch (active low)
 *
 *****/

#include <xc.h>
#include <stdint.h>

/***** CONFIGURATION *****/
// int reset, internal oscillator (no clock out), no watchdog timer
#pragma config MCLRE = OFF, FOSC = INTOSC, CLKOUTEN = OFF, WDTE = OFF
// brownout resets enabled, low brownout voltage, no low-power brownout reset
#pragma config BOREN = ON, BORV = LO, LPBOR = OFF
// no power-up timer, no code protect, no write protection
#pragma config PWRTE = OFF, CP = OFF, WRT = OFF
// stack resets on, high-voltage programming
#pragma config STVREN = ON, LVP = OFF

#define _XTAL_FREQ 500000           // oscillator frequency for _delay()

/***** MAIN PROGRAM *****/
void main()
{
    uint8_t    db_cnt;           // debounce counter

    /*** Initialisation

```

```

// configure port
LATA = 0; // start with all output pins low (LED off)
TRISA = ~(1<<1); // configure RA1 (only) as an output
OPTION_REGbits.nWPUEN = 0; // enable weak pull-ups (global)
WPUA = 1<<3; // enable pull-up on RA3 only

// configure oscillator
OSCCONbits.SCS1 = 1; // select internal clock
OSCCONbits.IRCF = 0b0111; // internal oscillator = 500 kHz

/** Main loop
for (;;)
{
    // wait for button press
    while (PORTAbits.RA3 == 1) // wait until RA3 low
        ;

    // toggle LED on RA1
    LATAbits.LATA1 = ~LATAbits.LATA1;

    // wait for button release, debounce by counting:
    for (db_cnt = 0; db_cnt <= 10; db_cnt++)
    {
        delay_ms(1); // sample every 1 ms
        if (PORTAbits.RA3 == 0) // if button down (RA3 low)
            db_cnt = 0; // restart count
    } // until button up for 10 successive reads
}
}

```

Conclusion

This lesson has shown that enhanced mid-range PICs are essentially similar to the mid-range devices we're familiar with, but are certainly "enhanced" in a number of useful ways.

In particular, we've seen that the enhanced mid-range architecture:

- provides data latch registers which avoid potential read-modify-write issues when modifying output pins
- has a default processor clock frequency, when using the internal RC oscillator, of 500 kHz instead of 4 MHz (although on the 12F1501 this is configurable)

Otherwise the facilities such as internal weak pull-ups are largely unchanged, but may be more flexible than before (weak pull-ups being available on every pin on the 12F1501, compared with not being available on GP3 on the 12F629).

Another facility that is largely unchanged in enhanced mid-range devices is the 8-bit Timer0 module, which we will take a brief look at in the next lesson.

Introduction to PIC Programming

Enhanced Mid-Range Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 1: Light an LED

This series of lessons introduces the enhanced mid-range PIC architecture, using assembly language. It starts by simply making an LED, connected to one of the output pins of a PIC, light up.

The apparently straightforward task of lighting an LED – never mind flashing it or anything else¹ – relies on:

- Having a functioning circuit in a workable prototyping environment
- Being able to use a development environment; to go from text to assembled PIC code
- Being able to correctly use a PIC programmer to load the code into the PIC chip
- Correctly configuring the PIC
- Writing code that will set the correct pin to output a high or low (depending on the circuit)

If you can get an LED to light up, then you know that you have a development, programming and prototyping environment that works, and enough understanding of the PIC architecture and instructions to get started. It's a firm base to build on.

In summary, this lesson covers:

- Introduction to the enhanced mid-range PIC architecture, using the PIC12F1501
- Simple control of digital output pins
- Using MPLAB X to create assembly language projects
- Using a PICKit 3 programmer with MPLAB X

Getting Started

For some background on PICs in general and details of the recommended development environment, see [lesson 0](#). Briefly, these tutorials assume that you are using a Microchip PICKit 3 programmer and either the [Gooligum Baseline and Mid-range PIC Training and Development Board](#) or Microchip's Low Pin Count (LPC) Demo Board, with Microchip's MPLAB X integrated development environment. But it is of course possible to adapt these instructions to different programmers and/or development boards.

The four LEDs on the LPC demo board don't work (directly) with 8-pin PICs, such as the 12F1501. So to complete this lesson, using an LPC demo board, you need to either add an additional LED and resistor to the prototyping area on your board, or use some solid core hook-up wire to patch one of the LEDs to the appropriate PIC pin, as described later.

¹ We'll get to the traditional first exercise in microcontroller programming of flashing an LED in [lesson 2](#)...

This is one reason the Gooligum training board was developed to accompany these tutorials – if you have the Gooligum board, you can simply plug in your 8-pin 12F PIC, and go.

We’re going to start with the simplest enhanced mid-range PIC – the PIC12F1501.

Of course, “simplest” is a relative term. The enhanced mid-range architecture is certainly more complex than the earlier baseline PIC architecture, introduced in the [Baseline PIC Assembler](#) tutorial series. Those lessons were able to start with a very simple PIC indeed (the 10F200), which made it possible to introduce only a few basic topics at first, without needing to say “ignore this for now; we’ll explain later”. More advanced topics were introduced by moving up to more advanced baseline and then eventually mid-range PICs through the [Mid-range PIC Assembler](#) tutorial series – building on what came before.

This tutorial series doesn’t refer back to those earlier lessons – it’s a fresh start. Unfortunately that does make it harder to ignore some of the complexities of the enhanced mid-range architecture, although we’ll keep it as simple as possible to begin with. If you do want to start learning with simpler PICs, you should consider working through the [baseline](#) and [mid-range](#) tutorial series.

But to repeat – the earlier lessons are not a prerequisite for these enhanced mid-range lessons.

In summary, for this lesson you should ideally have:

- A PC running Windows 7 or 8, with a spare USB port
- Microchip’s MPLAB X IDE software
- A Microchip PICkit 3 PIC programmer
- The Gooligum mid-range training board
- A PIC12F1501-I/P microcontroller (supplied with the Gooligum training board)

Introducing the PIC12F1501

When working with any microcontroller, you should always have on hand the latest version of the manufacturer’s data sheet. You should download the download the current data sheet for the 12F1501 from www.microchip.com.

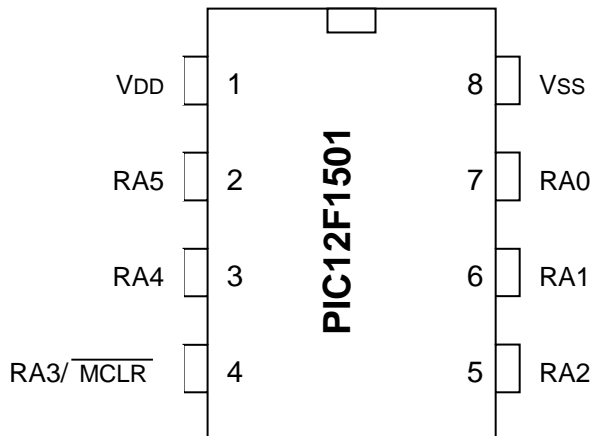
The features of various 8-pin PICs are summarised in the following table:

Device	Memory (words or bytes)			Timers		Analog		Clock rate (max MHz)
	Program	Data	EEPROM	8-bit	16-bit	Comp-arators	ADC inputs	
12F508	512	25	0	1	0	0	0	4
12F629	1024	64	128	1	1	1	0	20
12F675	1024	64	128	1	1	1	4	20
12F683	2048	128	256	2	1	1	4	20
12F1501	1024	64	0	2	1	1	4	20
12F1822	2048	128	256	2	1	1	4	32
12F1840	4096	256	256	2	1	1	4	32

We’ll look at the various features mentioned in the table, such as timers, analog inputs, and EEPROM memory, in later lessons. But even without knowing what these things are, you can see that the 12F1501 has fewer features than other enhanced mid-range PICs, such as the 12F1822 or 12F1840, while being roughly comparable to the 12F675 and 12F683 mid-range devices, and significantly more capable than the baseline 12F508.

The 12F family are all 8-pin devices, with six pins available for I/O (input and output).

They share a common pin-out, as shown below.



VDD is the positive power supply.

VSS is the “negative” supply, or ground. All of the input and output levels are measured relative to VSS. In most circuits, there is only a single ground reference, considered to be at 0 V (zero volts), and VSS will be connected to ground.

The power supply voltage on the PIC12F1501 can range from 2.3 V to 5.5 V².

This wide range means that the PIC’s power supply can be very simple. Depending on the circuit, you may need no more than a pair of 1.5 V batteries.

Normally you’d place a capacitor, typically 100 nF and ceramic, between VDD and VSS, close to the chip, to smooth transient changes to the power supply voltage caused by changing loads (e.g. motors, or something as simple as an LED turning on) or noise in the circuit.

The remaining pins, RA0 to RA5, are the I/O pins. They are used for digital input and output, except for RA3, which can only be an input. The other pins – RA0, RA1, RA2, RA4 and RA5 – can be individually set to be inputs or outputs.

PIC12F1501 Internals

8-bit PICs use a so-called Harvard architecture, where program and data memory is entirely separate.

In the 12F1501, program memory extends from 0000h to 03FFh (hexadecimal). Each of these 1024 addresses can hold a separate 14-bit *word* – usually a program instruction.

User code starts by executing the instruction at 0000h, and then proceeds sequentially from there – unless of course your program includes loops, branches or subroutines, which any real program will!

The data memory, also known as the *register file*, in enhanced mid-range PICs is *banked*; it is divided into 32 banks, with 128 addresses in each bank. Each address may be empty, or it may hold an 8-bit *register* used to control the PIC or a byte of general purpose memory where your program can store data.

Each bank (other than bank 31, as we’ll see later) is laid out the same way, as illustrated on the right.

Standard bank layout

Offset	Register Type
00h	Core Registers
0Bh	
0Ch	Special Function Registers
1Fh	
20h	General Purpose RAM
6Fh	
70h	
7Fh	Common RAM

² A low-power variant, the PIC12LF1501, is also available, where VDD can range from 1.8 V to 3.6 V, with at least 2.5 V needed for clock rates above 16 MHz in both variants.

Core Registers

The first 12 addresses of every bank provide access to the “core registers”, as shown on the right.

There is only one set of core registers, but they appear in the same locations within each bank – meaning that they can be accessed in the same way, regardless of which bank is selected.

That’s useful, because these are the registers that most directly affect the device’s basic operation – you’ll need to access some of them very often, so it’s very helpful to have them always available.

Instead of attempting to explain what each of these registers is used for, we’ll leave most of those explanations until later, when the topics they relate to are introduced.

But a couple are worth mentioning now.

Core Registers

Offset	Register Name
00h	INDF0
01h	INDF1
02h	PCL
03h	STATUS
04h	FSR0L
05h	FSR0H
06h	FSR1L
07h	FSR1H
08h	BSR
09h	WREG
0Ah	PCLATH
0Bh	INTCON

The working register (*W* and *WREG*)

8-bit PICs use a “working register”, usually referred to as ‘*W*’, which is central to their operation. It’s the equivalent of the ‘accumulator’ in some other microprocessors. For example, to copy data from one register to another, you have to copy it into *W* first, and then copy from *W* to the destination. Or, to add two numbers, one of them has to be in *W*. *W* is used a lot!

Many PIC instructions implicitly access or operate on *W*.

For example, to load the value ‘5’ into *W*, you would use:

```
movlw    5
```

‘`movlw`’ is our first PIC assembler instruction. It loads the *W* register with an 8-bit value (between 0 and 255), which may represent a number, character, or something else.

Microchip calls a value like this, which is embedded in an instruction, a *literal*. They also refer to a load or store operation as a ‘move’ (even though nothing is moved; the source never changes).

So, ‘`movlw`’ means “**move** literal to **W**”.

If you wanted to write that value into (for example) the *BSR* register, you would use:

```
movwf    BSR
```

The ‘`movwf`’ instruction copies (Microchip would say “moves”) the contents of the *W* register into the specified register – “**move W** to register file”.

The *WREG* register provides access to *W*, for those occasional situations where it would be useful to be able to explicitly operate on *W*.

We’ll see some examples of using *WREG* in later lessons, but most of the time you’ll use instructions such as “`movlw`” or “`movwf`” to access *W* directly; it’s rare to have to access it via the *WREG* register.

Banking and the bank select register (BSR)

At the lowest level, PIC instructions consist of bits. In the enhanced mid-range PIC architecture, each instruction word is fourteen bits wide.

Some of these bits designate which instruction it is; this set of bits is called the *opcode*.

For example, the opcode for `movlw` is 110000.

The remaining bits in each 14-bit instruction word are used to specify whatever value is associated with that instruction, such as a literal value or a register address. In the case of `movlw`, the opcode is six bits long, leaving the other eight bits to hold the literal value that will be moved into *W*. Thus, the 14-bit instruction word for '`movlw 1`' is 110000 00000001 in binary, the first six bits meaning '`movlw`' and the last eight bits being the binary for '1'.

In the enhanced mid-range architecture, only seven bits are allocated to register addressing.

For example, the opcode for `movwf` is 0000001, which is seven bits long, and the remaining seven bits specify which register is to be acted on (the register that the contents of *W* will be copied into).

Seven bits is enough to allow up to 128 registers, numbered from 0 to 127 (or 00h to 7Fh in hexadecimal), to be addressed. This 7-bit register addressing limitation is why each *register bank* consists of only 128 addresses. So to allow for more than a total of 128 registers, or data memory addresses, the data memory has to be divided into multiple banks – 32 of them in the enhanced mid-range architecture.

This scheme provides for $32 \times 128 = 4096$ data memory addresses.

To specify which of these 4096 possible addresses (000h – FFFh) an instruction will access, we need two things: the address *offset* (00h – 7Fh) within the bank, and which bank (0 – 31 or 00h – 1Fh) to access.

As we've seen, the offset is specified as a 7-bit field within the instruction opcode. But how to specify the bank to access, if it's not part of the instruction? That's where the bank select register, **BSR**, comes in.

As you might guess, **BSR** holds a 5-bit number (0 – 31) specifying the currently selected bank.

When an instruction accesses a register, the 7-bit offset is taken from the instruction opcode, while the bank number is taken from **BSR**. Or putting it another way, **BSR** provides the most significant five bits of the 12-bit data memory address, with the instruction opcode providing the lower 7 bits of the address.

When you're accessing a core register, banking isn't a problem. The core registers appear in every bank, always at the same offset, so it doesn't matter which bank is selected when you go to access them.

But as we'll see in the next section, the special function registers are different in each bank, making it necessary to select the correct bank before attempting to access any of them.

For example, the **TRISA** register, which we'll introduce later in this lesson, is at data memory address 08Ch, which is another way of saying that it's at address offset 0Ch in bank 1.

Suppose we want to write the binary value '111101' into **TRISA** (you'll why, in the example later).

To access **TRISA**, we must first select bank 1, by writing the value '1' into **BSR**.

We can do this with:

```
movlw    1                ; select bank 1:
movwf    BSR              ; write '1' to BSR
```

(note that comments in PIC assembly language begin with a '`;`')

Alternatively (and better), we could use an enhanced mid-range PIC instruction, ‘movlb’, which writes a literal value directly into the bank select register – “**move literal to BSR**”

For example:

```
movlb    1            ; select bank 1
```

We’re then free to access TRISA:

```
movlw    b'111101'    ; write binary '111101'
movwf    TRISA        ;    to TRISA
```

(note the “b'111101'” syntax, used to specify a binary value)

By the way, it should be obvious why BSR has to be available, at the same location, in every bank. If BSR only appeared in one bank, you’d have to select that bank before you could update BSR – but to select a bank, you need to update BSR... Having BSR available in every bank avoids this “Catch 22” situation.

The banksel directive

Of course, it’s a bit painful, and potentially quite error-prone, to have to figure out which bank each register is in, and load the appropriate value into BSR, before you can access that register.

But luckily the assembler provides a *directive*, `banksel`, which automates the process.

To use it, you simply specify which register you wish to access, and the assembler then generates the correct bank selection code for us.

For example, our fragment of code to load ‘111101’ into TRISA becomes:

```
banksel  TRISA        ; select the bank containing TRISA
movlw    b'111101'    ; write binary '111101'
movwf    TRISA        ;    to TRISA
```

Special Function Registers

The next 20 addresses in banks 0 to 30 hold the special function registers (SFRs) used to access and control the device’s features, including ports (basic digital input and output pins) and peripherals such as timers, comparators, analog-to-digital converters and pulse-width modulated (PWM) outputs – topics we’ll look at in later lessons.

With $20 \times 31 = 620$ addresses available for SFRs, the enhanced mid-range architecture has enough potential register locations to support a wide range of advanced peripherals, such as USB, which require a large number of registers to configure, control and access them.

But since the 12F1501 is a relatively basic device, compared with other enhanced mid-range PICs, it doesn’t need hundreds of special function registers. Many of its 32 banks do not contain any SFRs. Others contain only a few.

Nevertheless, the 12F1501 does have dozens of special function registers, scattered across several banks.

It doesn’t make sense to list them all here – you should refer to the PIC12F1501 data sheet for that – but the table on the next page shows the content of the first four banks, where most of the commonly-used SFRs, including those references in this lesson, are located.

PIC12F1501 Special Function Registers (banks 0 – 3)

Bank 0		Bank 1		Bank 2		Bank 3			
000h	Core Registers	080h	Core Registers	100h	Core Registers	180h	Core Registers		
00Bh		08Bh		10Bh		18Bh			
00Ch	PORTA	08Ch	TRISA	10Ch	LATA	18Ch	ANSELA		
00Dh		08Dh		10Dh		18Dh			
00Eh		08Eh		10Eh		18Eh			
00Fh		08Fh		10Fh		18Fh			
010h		090h		110h		190h			
011h	PIR1	091h	PIE1	111h	CM1CON0	191h	PMADRL		
012h	PIR2	092h	PIE2	112h	CM1CON1	192h	PMADRH		
013h	PIR3	093h	PIE3	113h		193h	PMDATL		
014h		094h		114h		194h	PMDATH		
015h	TMR0	095h	OPTION_REG	115h	CMOUT	195h	PMCON1		
016h	TMR1L	096h	PCON	116h	BORCON	196h	PMCON2		
017h	TMR1H	097h	WDTCON	117h	FVRCON	197h	VREGCON		
018h	T1CON	098h		118h	DACCON0	198h			
019h	TMR2	099h	OSCCON	119h	DACCON1	199h			
01Ah	PR2	09Ah	OSCSTAT	11Ah		19Ah			
01Bh	T2CON	09Bh	ADRESL	11Bh		19Bh			
01Ch		09Ch	ADRESH	11Ch		19Ch			
01Dh		09Dh	ADCON0	11Dh	APFCON	19Dh			
01Eh		09Eh	ADCON1	11Eh		19Eh			
01Fh		09Fh	ADCON2	11Fh		19Fh			
020h	General Purpose RAM	0A0h		120h		1A0h			
04Fh									
050h									
06Fh				0EFh				16Fh	
070h	Common RAM	0F0h	Common RAM	170h	Common RAM	1F0h	Common RAM		
07Fh		0FFh		17Fh		1FFh			

General Purpose RAM

The next 80 addresses in banks 0 to 30 are available for “general purpose RAM”

Also known as a “general purpose registers”, or GPRs, these data memory locations are available for use by your program to store its variables and other data³.

These registers are banked – you need to select the appropriate bank (by writing the bank number into BSR, via `movlb` or `banksel`, in the same way as for SFRs) before accessing general purpose RAM.

³ Note however that their contents are lost whenever the device loses power.

This means that enhanced mid-range PICs can have up to $80 \times 31 = 2480$ bytes of general purpose RAM.

The 12F1501, however, has only 48 bytes of general purpose RAM, all in bank 0, as shown in the diagram on the previous page.

So, although the enhanced mid-range architecture allows for us to 2480 bytes of general purpose RAM, most devices will have less than that amount.

Common RAM

The last 16 addresses of each bank are used for “common RAM”.

Every enhanced mid-range PIC device has exactly 16 bytes of common RAM – no more, and no less.

It is equivalent to general purpose RAM, except it is mapped into the same location in every bank.

This makes it very useful for the storage of commonly-accessed variables, allowing you to reduce your code size by avoiding the need for bank selection instructions when accessing those variables. But with only 16 bytes available, you should try to use it sparingly.

So overall the 12F1501 has only 48 bytes of general purpose RAM, all in bank 0, along with 16 bytes of common RAM (mapped into every bank) giving it a total of 64 bytes of data memory.

Finally, bank 31 contains registers related to interrupts and the hardware stack – topics that, once again, we’ll look at in future lessons.

PIC12F1501 Input and Output

As mentioned above, the 12F1501 has six I/O pins: RA0, RA1, RA2, RA4 and RA5, which can be used for digital input and output, plus RA3, which is input-only.

Taken together, the six I/O pins comprise the general-purpose I/O *port*, or GPIO port, which is referred to as PORTA on the 12F1501.

The PORTA register provides access to the port pins:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTA			RA5	RA4	RA3	RA2	RA1	RA0

If a pin is configured as an output, setting the corresponding PORTA bit to ‘1’ outputs a high voltage⁴ on the corresponding pin; clearing it to ‘0’ outputs a low voltage⁵.

Reading the PORTA register reads the voltage present on each pin. If the voltage on a pin is high⁶, the corresponding bit reads as ‘1’; if the input voltage is low⁷, the corresponding bit reads as ‘0’.

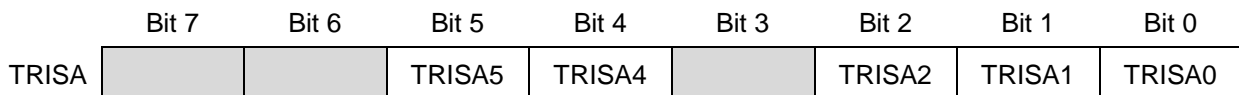
⁴ a ‘high’ output will be within 0.7 V of the supply voltage (VDD), for small pin currents (< 3.5 mA with VDD = 5 V)

⁵ a ‘low’ output is less than 0.6 V, for small pin currents (< 8 mA with VDD = 5 V)

⁶ the threshold level depends on the power supply, but a ‘high’ input is any voltage above 2.0 V, given a 5 V supply

⁷ a ‘low’ input is anything below 0.8 V, given a 5 V supply – see the data sheet for details of each of these levels

The TRISA register controls whether a pin is configured as an input or output:



To configure a pin as an input, set the corresponding bit in the TRISA register to '1'. To make it an output, clear the corresponding TRISA bit to '0'.

Why is it called 'TRIS'? Each pin (except RA3) can be configured as one of three states: high-impedance input, output high, or output low. In the input state, the PIC's output drivers are effectively disconnected from the pin. Another name for an output that can be disconnected is '*tri-state*' – hence, TRIS.

Note that bit 3 of TRISA is greyed-out. Clearing this bit will have no effect, as RA3 is always an input.

The default state for each pin is 'input'; TRIS is set to all '1's when the PIC is powered on or reset.

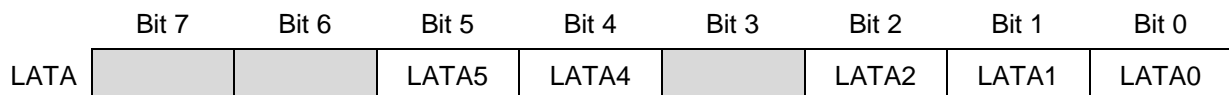
It's important to understand that, regardless of whether a pin is configured as an input or an output, PORTA reflects the actual voltage present on that pin.

Note: the port registers represent the actual voltages present on each digital I/O pin, including pins configured as digital outputs

If you attempt to output a 'high' on an output pin by writing a '1' to the corresponding port bit, but the external circuit holds that pin low, that pin will read as '0' – not what you might have expected.

This behaviour can lead to what are known as *read-modify-write* problems, where instructions which are intended to modify only specific pins actually read the entire port, including pins which may not reflect the value that had been output to them, and then write the new value (with some bits possibly incorrect) back to the port.

To avoid the potential for read-modify-write problems, the enhanced mid-range architecture makes available an "output data latch" register, associated with each port:



Writing to LATA has the same effect as writing to PORTA: if a pin is configured as an output, setting the corresponding LATA bit to '1' outputs a 'high' on that pin; clearing it to '0' outputs a 'low'.

However – reading LATA returns the value that was last written to LATA. It does not read the voltages on the pins (whether input or output) themselves.

This means that you can avoid read-modify-write problems by following these rules:

- if you are writing an entire byte to a port, you can write to either PORTA or LATA
- if you are modifying individual port pins, you should operate on LATA
- if you are reading digital input pins, you must read PORTA

To keep it simpler, you won't run into any problems if you always access LATA to write to or modify output pins, and PORTA to read digital input pins.

This should become clearer as we work through examples.

When configured as an output, each I/O pin on the 12F1501 can source or sink (i.e. current into or out of the pin) up to 25 mA – enough to directly drive an LED.

PICs are tough devices, and you may get away with exceeding these limits – but if you ignore the absolute maximum ratings specified in the data sheet, you’re on your own. Maybe your circuit will work, maybe not. Or maybe it will work for a short time, before failing. It’s better to follow the data sheet...

Example Circuit

We now have enough background information to design a circuit to light an LED.

We’ll need a regulated power supply, let’s assume 5 V, connected to VDD and VSS. And remember that we should add a bypass capacitor, preferably a 100 nF (or larger) ceramic, across it.

We’ll also need an LED of course, and a resistor to limit the current.

Although the PIC12F1501 can supply up to 25 mA from a single pin, 10 mA is more than enough to adequately light most LEDs. With a 5 V supply and assuming a red or green LED with a forward voltage of around 2 V, the voltage drop across the resistor will be around 3 V.

Applying Ohm’s law, $R = V / I = 3 \text{ V} \div 10 \text{ mA} = 300 \Omega$. Since precision isn’t needed here (we only need “about” 10 mA), it’s ok to choose the next highest “standard” E12 resistor value, which is 330 Ω . It means that the LED will draw less than 10 mA, but that’s a good thing, because, if we’re going to use a PICKit 3 to power the circuit, we need to limit overall current consumption to 30 mA, because that is the maximum current the PICKit 3 can supply.

Finally, we need to connect the LED to one of the PIC’s pins.

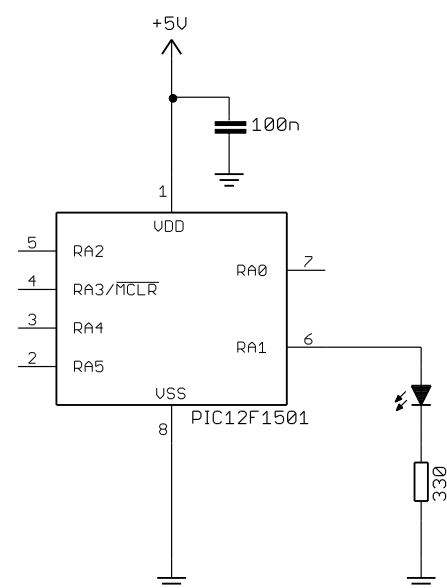
We can’t choose RA3, because it’s input-only.

If you’re using the Gooligum training board, you could choose any of the other pins, but if you use the Microchip LPC Demo Board to implement the circuit, it’s not a good idea to use RA0, because it’s connected to a trimpot on the LPC demo board, which would divert current from the LED.

So, we’ll use RA1, giving the circuit shown on the right.

Simple, isn’t it? Modern microcontrollers really do have minimal requirements.

Of course, some connections are also needed for the ICSP (programmer) signals. These will be provided by your development board, unless you are building the circuit yourself. But the circuit as shown here is all that is needed for the PIC to run, and light the LED.



Gooligum training and development board instructions

If you have the Gooligum training board, you can use it to implement this circuit.

Plug the PIC12F1501 into the top section of the 14-pin IC socket – the section marked ‘12F’⁸.

Connect a shunt across the jumper (JP12) on the LED labelled ‘RA1’, and ensure that every other jumper is disconnected.

Plug your PICKit 3 programmer into the ICSP connector on the training board, with the arrow on the board aligned with the arrow on the PICKit, and plug the PICKit into a USB port on your PC.

⁸ Note that, although the PIC12F1501 comes in an 8-pin package, **it will not work** in the 8-pin ‘10F’ socket. You must install it in the ‘12F’ section of the 14-pin socket.

The PICkit 3 can supply enough power for this circuit, so there is no need to connect an external power supply.

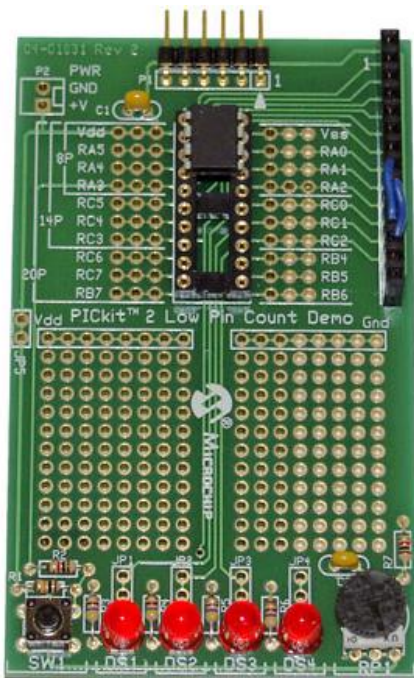
Microchip Low Pin Count Demo Board instructions

If you are using Microchip's LPC Demo Board, you'll need to take some additional steps.

Although the board provides four LEDs, they cannot be used directly with a 12F1501 (or any 8-pin PIC), because those LEDs are connected to DIP socket pins which are only used with 14-pin and 20-pin devices.

However, the circuit can be readily built by adding an LED, a 330 Ω resistor and a piece of wire to the LPC Demo Board, as illustrated on the right.

In the pictured board, a green LED is wired to RA1 and a red LED to RA2; we'll use both LEDs in later lessons. Jumper blocks have been added so that these LEDs can be easily disconnected from the PIC, to facilitate prototyping other circuits. These jumpers are wired in series with each LED.



If you prefer not to solder components onto your demo board, you can use the LEDs on the board, labelled 'DS1' to 'DS4', by making connections on the 14-pin header on the right of the demo board, as shown on the left. This header makes available all the 12F1501's pins, RA0 – RA5, as well as power (+5 V) and ground. It also brings out the additional pins, labelled 'RC0' to 'RC5', available on 14-pin PIC devices.

The LEDs are connected to the pins labelled 'RC0' to 'RC3' on the IC socket, via 470 Ω resistors (and jumpers, if you choose to install them). 'DS1' connects to pin 'RC0', 'DS2' to 'RC1', and so on.

So, to connect LED 'DS2' to pin RA1, simply connect the pin labelled 'RA1' to the pin labelled 'RC1', which can be done by plugging a short piece of solid-core hook-up wire between pins 8 and 11 on the 14-pin header.

Similarly, to connect LED 'DS3' to pin RA2, simply connect header pins 9 and 12.

That's certainly much easier than soldering, so why bother adding LEDs to the demo board? The only real advantage is that, when using 14-pin and 20-pin PICs later, you may find it useful to have LEDs available on RA1 and RA2, while leaving RC0 – RC3 available to use, independently. In any case, it is useful to leave the 14-pin header free for use as an expansion connector, to allow you to build more complex circuits, such as those found in the later tutorial lessons.

Time to move on to programming!

Development Environment

You'll need Microchip's MPLAB Integrated Development Environment (MPLAB IDE), which you can download from www.microchip.com.

As discussed in [lesson 0](#), MPLAB comes in two varieties: the older, Windows-only MPLAB 8, and the newer multi-platform MPLAB X. Although MPLAB 8 remains a stable and usable environment, it has been effectively retired by Microchip and will not continue to be updated.

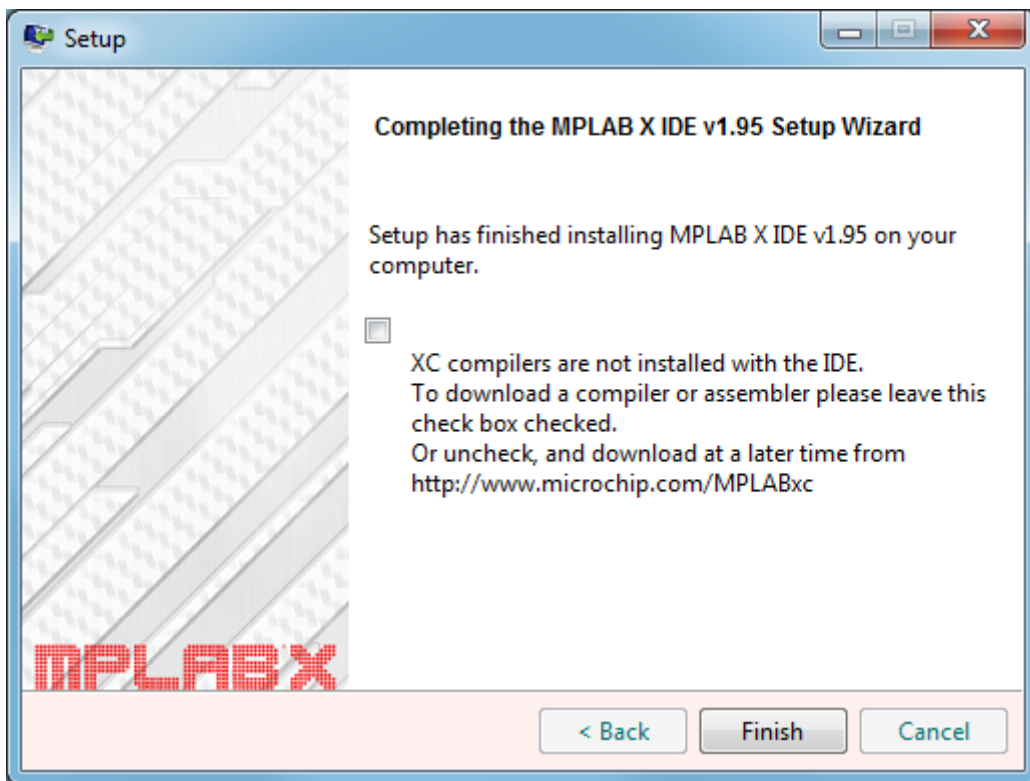
Therefore, this tutorial series assumes that you will be using MPLAB X.

Installation

You should download the MPLAB X IDE installer for your platform (Windows, Linux or Mac) from the MPLAB X download page at www.microchip.com, and then run it.

There are no installation options (other than being able to choose the installation directory). It's an "all or nothing" installer, including the MPASM assembler and support for all of Microchip's PIC MCUs and development tools.

When the installation completes, you are prompted to download and one (or more) of Microchip's "XC" series of C compilers:



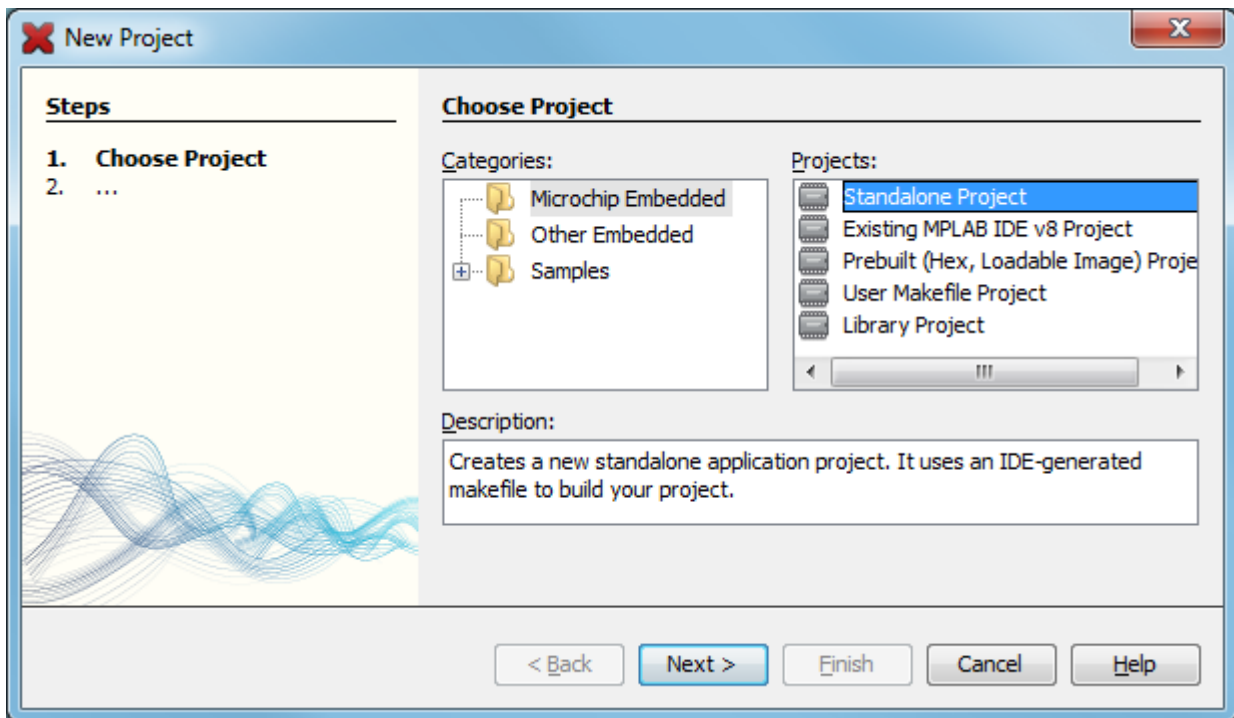
Even if you want to use C in addition to assembly language, there is no problem with downloading and installing the compilers separately, later. So you can leave this box unchecked for now, and then click 'Finish' to finish the installation.

Creating a New Project

When you first run MPLAB X, you will see the "Learn & Discover" tab, on the Start Page.

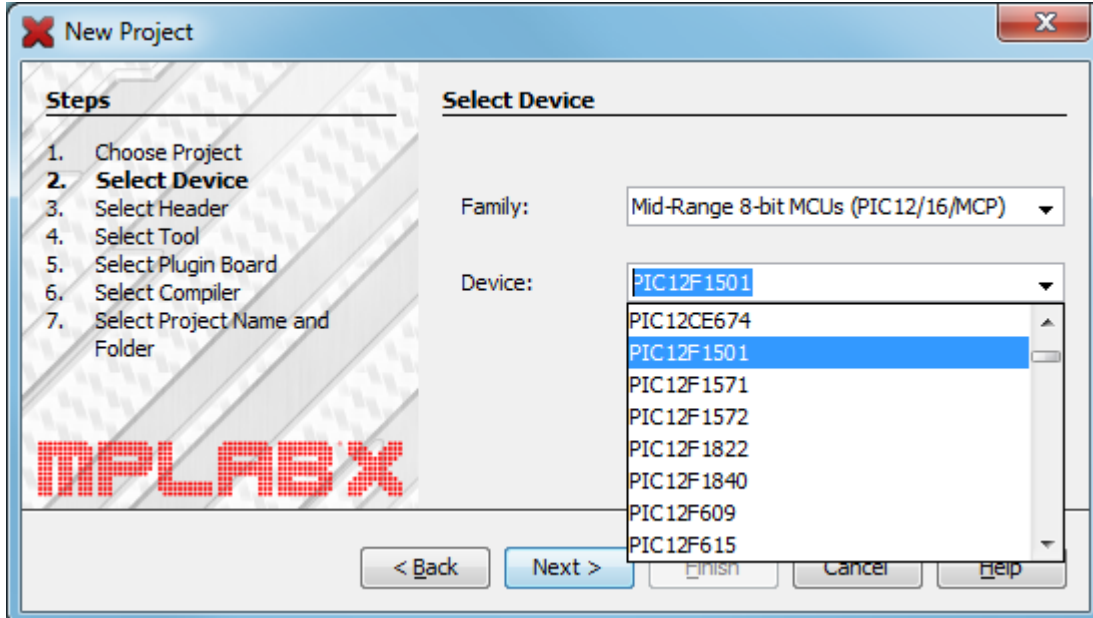
To start a new project, you should run the New Project wizard, by clicking on 'Create New Project'.

In the first step, you need to specify the project category. Choose ‘Standalone Project’:



Next, select the PIC family and device.

In our case, we need ‘Mid-Range 8-bit MCUs’ as the family, and ‘PIC12F1501’ as the device:

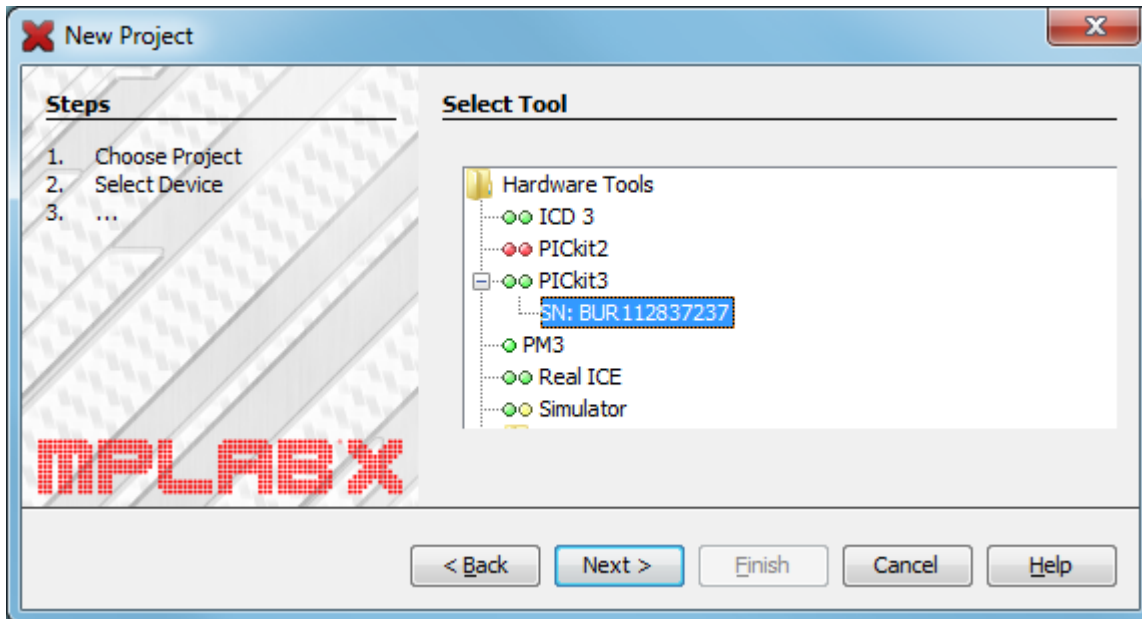


The third step allows you to optionally select a debug header.

This is a device used to facilitate hardware debugging (see explanation in [lesson 0](#)), especially for PICs (such as the 12F1501) which do not include internal hardware to support debugging. If you are just starting out, you are unlikely to have one of these debug headers, and you don’t need one for these tutorials. So, you should not select a header. Just click ‘Next’.

The next step is to select the tool you will use to program your PIC.

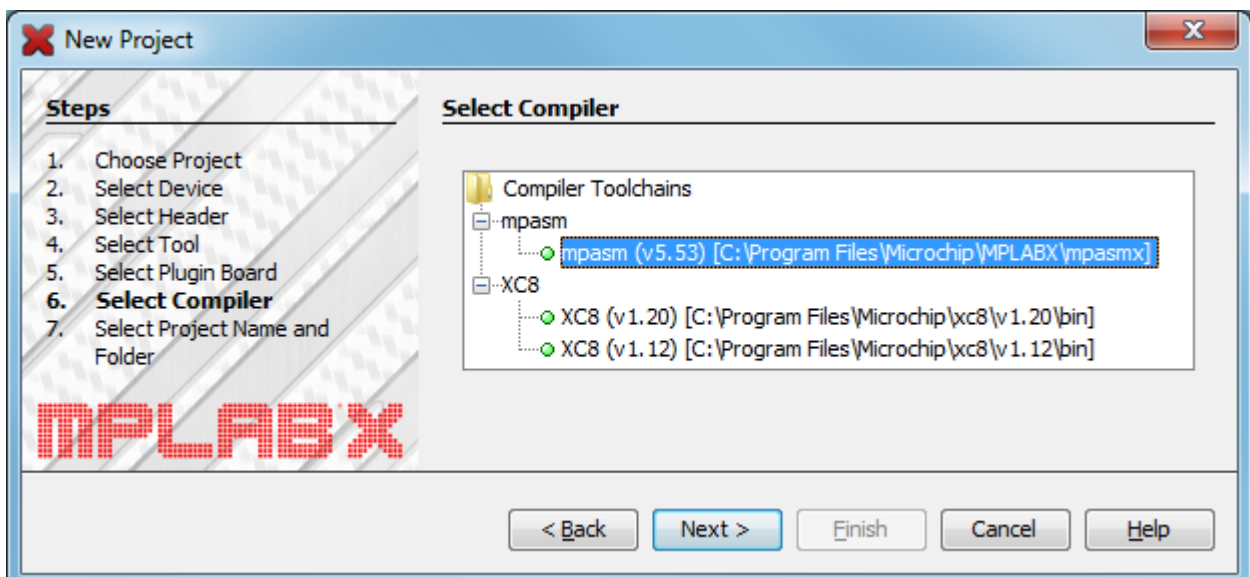
First, you should plug in the programmer (e.g. PICKit 3) you intend to use. If it is properly connected to your PC, with a functioning device driver⁹, it will appear in the list of hardware tools, and you should select it, as shown:



In this case, a PICKit 3 is connected to the PC.

If you have more than one programmer plugged in (including more than one of the same type, such as two PICKit 3s), they will all appear in this list, and you should select the specific one you intend to use for this project – you may need to check the serial number. Of course, you probably only have one programmer, so your selection will be easy.

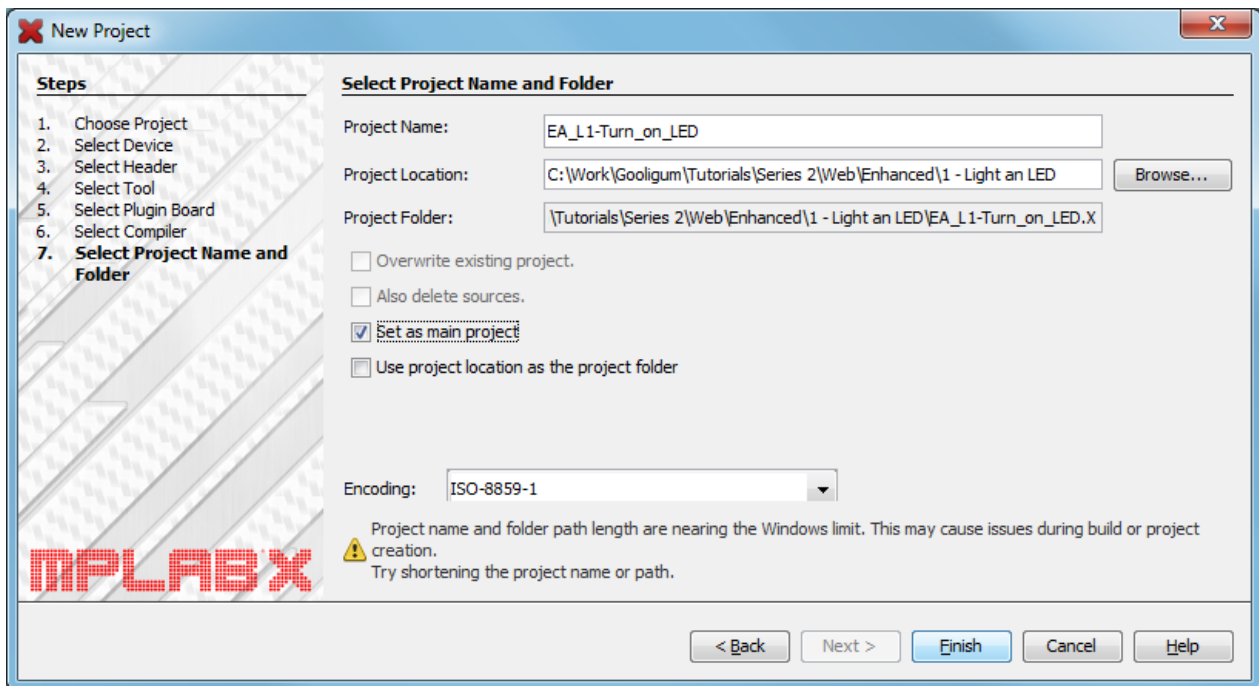
After selecting the hardware tool, you select the compiler (or, in our case, assembler) you wish to use:



To specify that we will be programming in assembler, select the ‘mpasm’ option.

⁹ There is no need to install a special device driver for the PICKit 3; it works “out of the box”.

Finally, you need to specify your project's location, and give it a name:



For example, in the environment used to develop these tutorials, all the files related to this lesson, including schematics and documentation, are placed in a folder named '1 - Light an LED', which is the "Project Location" given above.

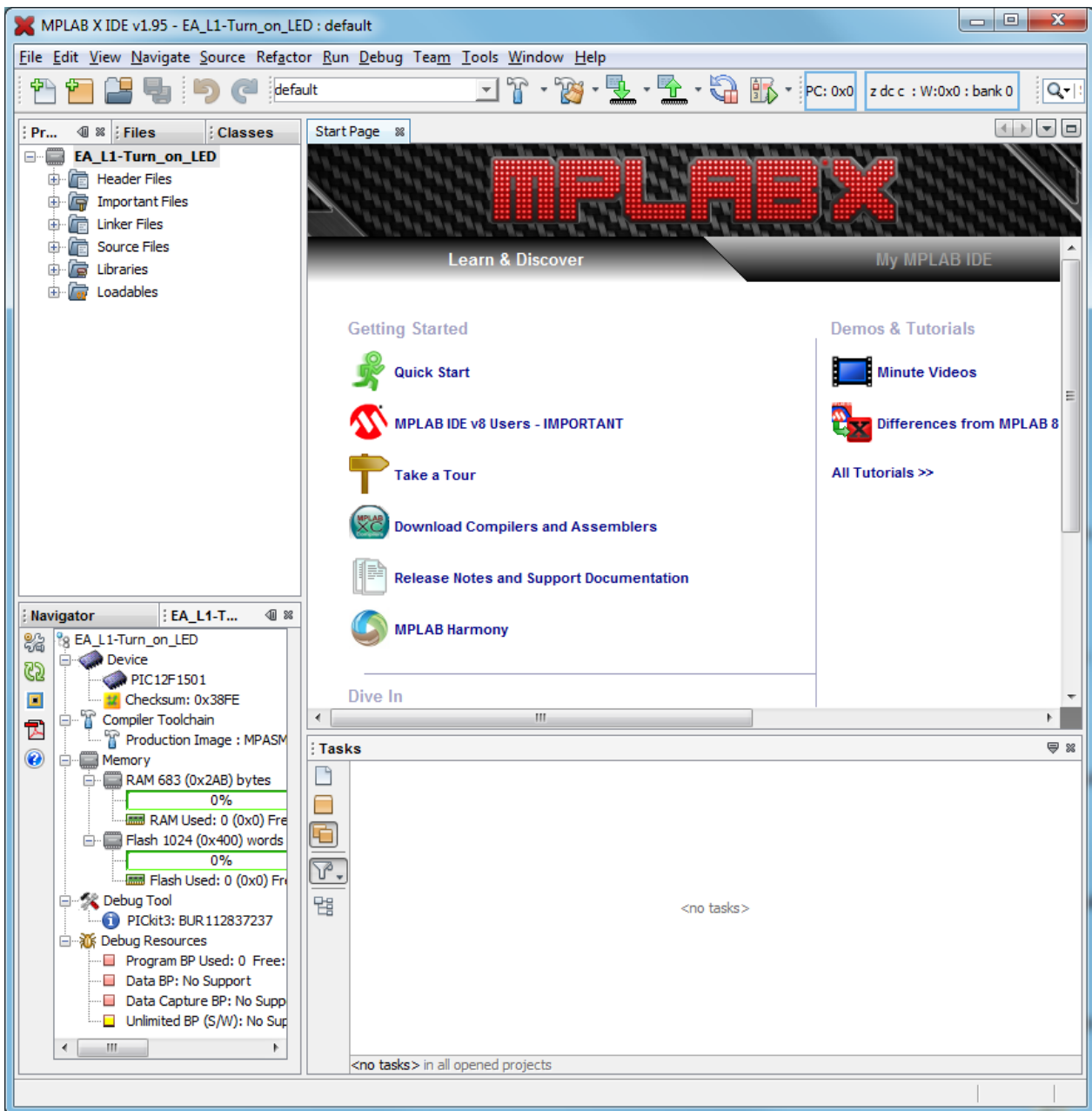
By default, MPLAB X then creates a separate folder for the PIC source code and other files related to this project, in a subfolder that has the same name as the project, with a '.X' on the end. If you wish, you can remove the '.X' extension from the project folder, before you click on 'Finish'.

If you select "Use project location as the project folder", this behaviour changes – the project files are then placed in the "Project Location" folder, instead of being in a separate folder. This isn't recommended, which is why that box is left unchecked above. But if you prefer not to have a separate folder for the MPLAB files, you can select this option.

Note the warning about project name and folder path length. To avoid possible problems, it's best to use shorter names and paths, when using Windows, although in this case it's actually ok.

Since this is the only project we're working on, it doesn't make much difference whether you select "Set as main project"; this is something that is more useful when you are working with multiple projects.

After you click “Finish”, your workspace should look something like this:



The panel in the top left allows you to see and access the various files that comprise your project – we’ll add a source code file (which will appear under “Source Files” in the next step).

The panel in the bottom left shows your project’s configuration and status, such as which device you’re using (12F1501), the selected programmer (shown here as “Debug Tool”) and the amount of PIC program (“Flash”) and data (“RAM”) memory our program is using – 0% for now, because we haven’t created a program yet!

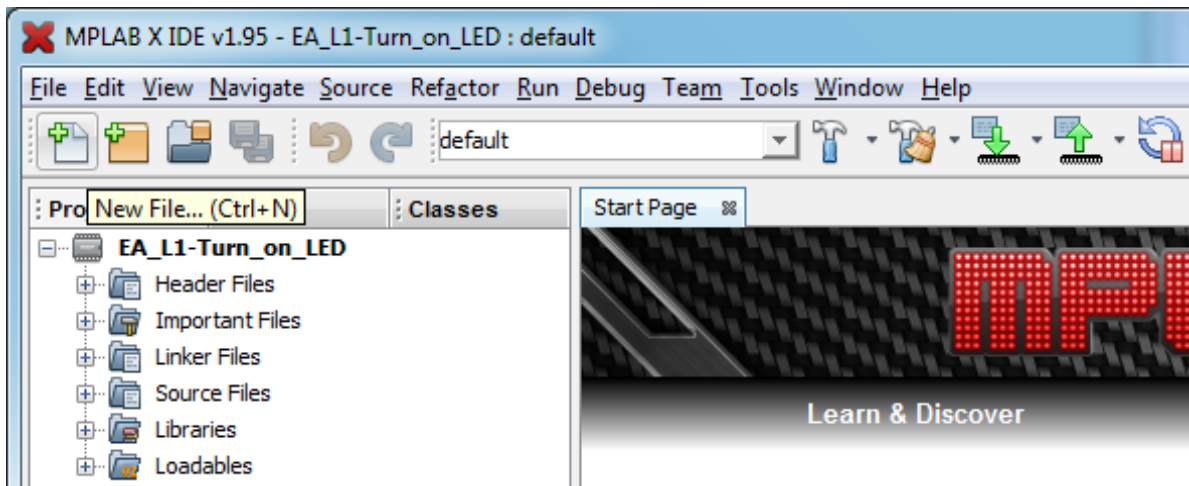
The largest panel, in the upper right, is where you edit your source code.

Below it is a panel where you’ll see the status of processes such as assembling your program and programming the PIC.

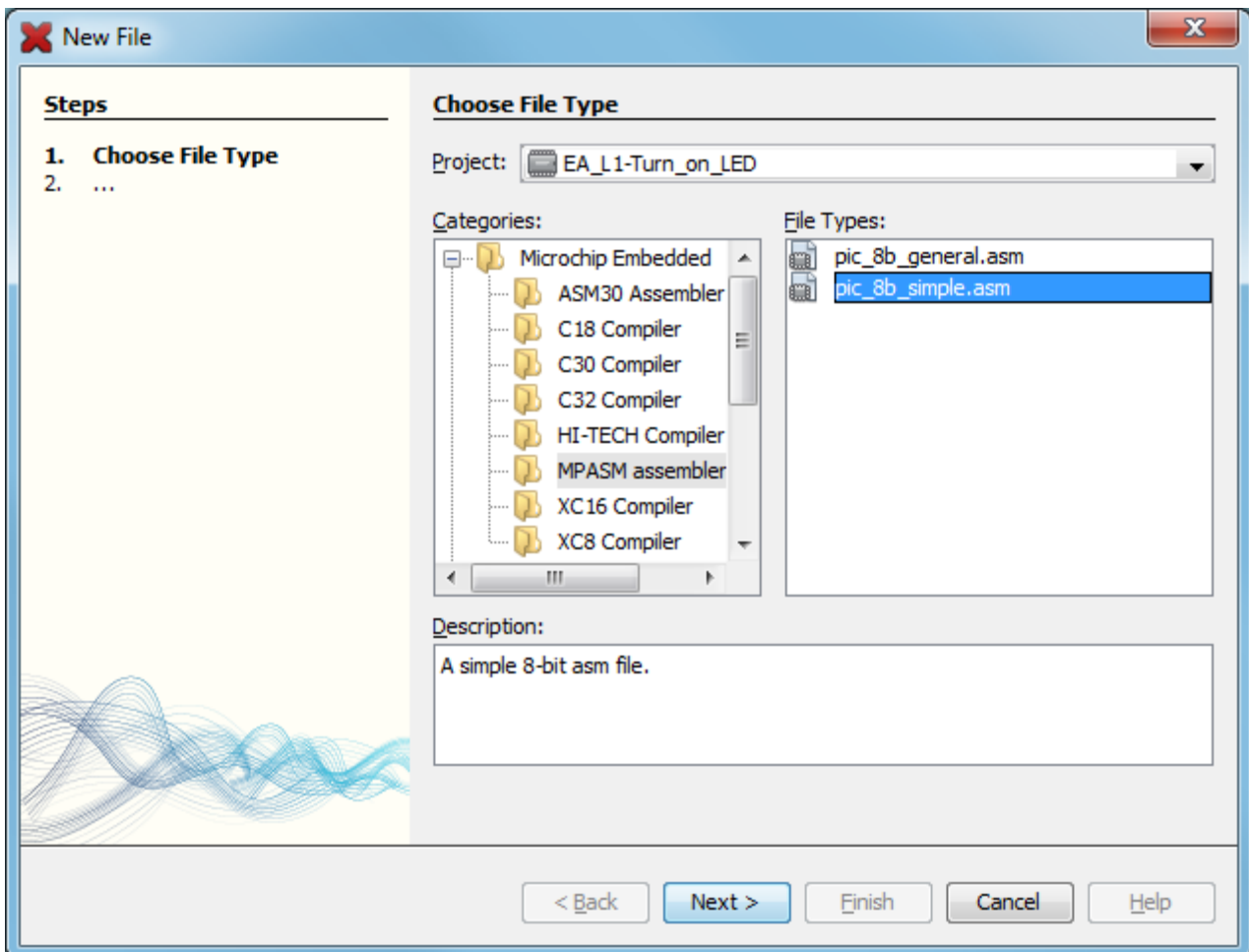
Note that MPLAB X has many, many features that we won’t be exploring in these tutorials. These lessons are about PIC programming, not using MPLAB X. So it’s worth taking some time to explore the training resources available from the “Learn & Discover” tab shown above.

There are a couple of ways to create a new source file and add it to the project.

You could select the “File → New File...” menu item, press Ctrl+N, or click on the “New File” button in the toolbar:



This will open the New File window:

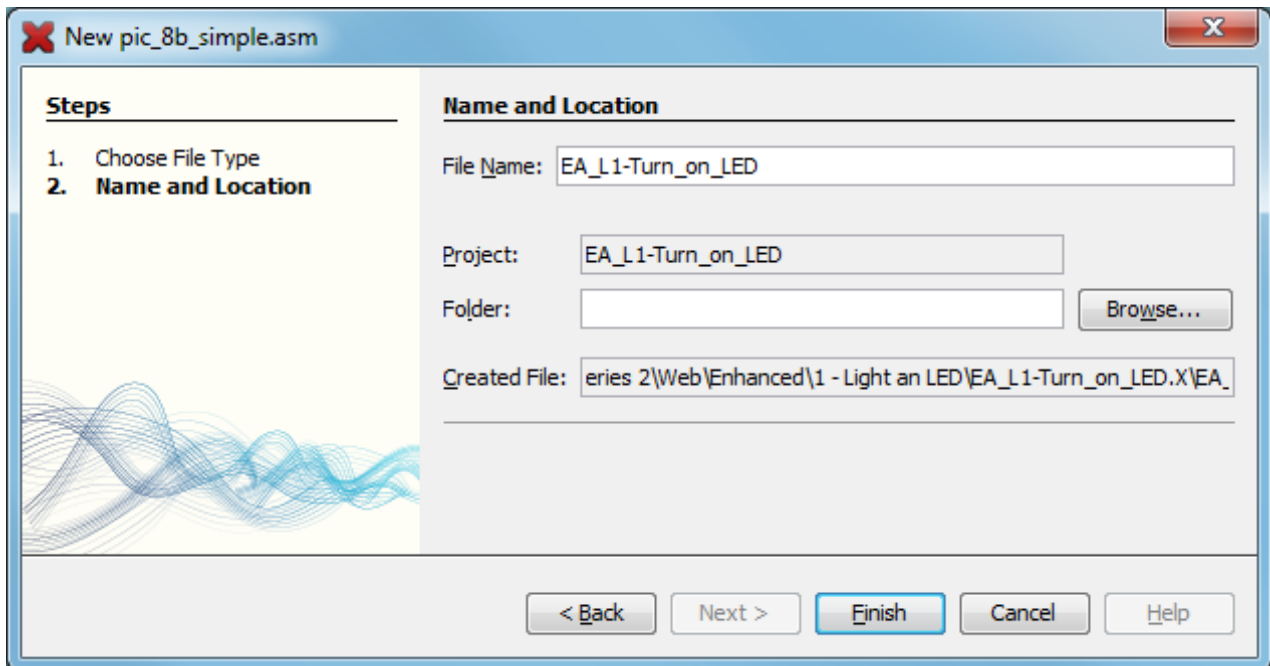


Microchip provide a number of templates to base your source file on.

We’re creating an MPASM assembler project, so navigate to “MPASM assembler”, within the “Microchip Embedded” category, then select “pic_8b_simple.asm”, as shown.

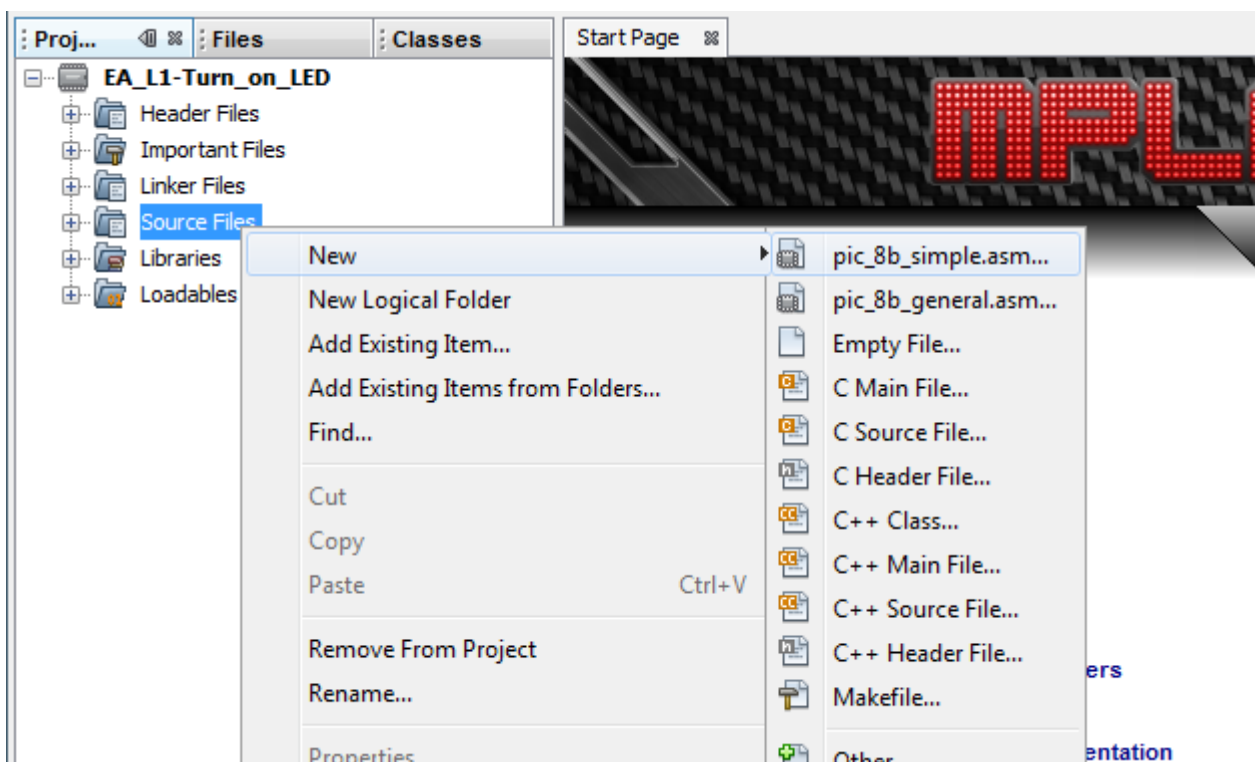
The “8b” refers to 8-bit PICs. Since we’re starting with a simple example, the “simple 8-bit asm file” is the best choice for now. But when you begin working on more advanced programs, you may find that the “pic_8b_general.asm” option is a better choice.

After you click ‘Next’, you have the option of naming your file:

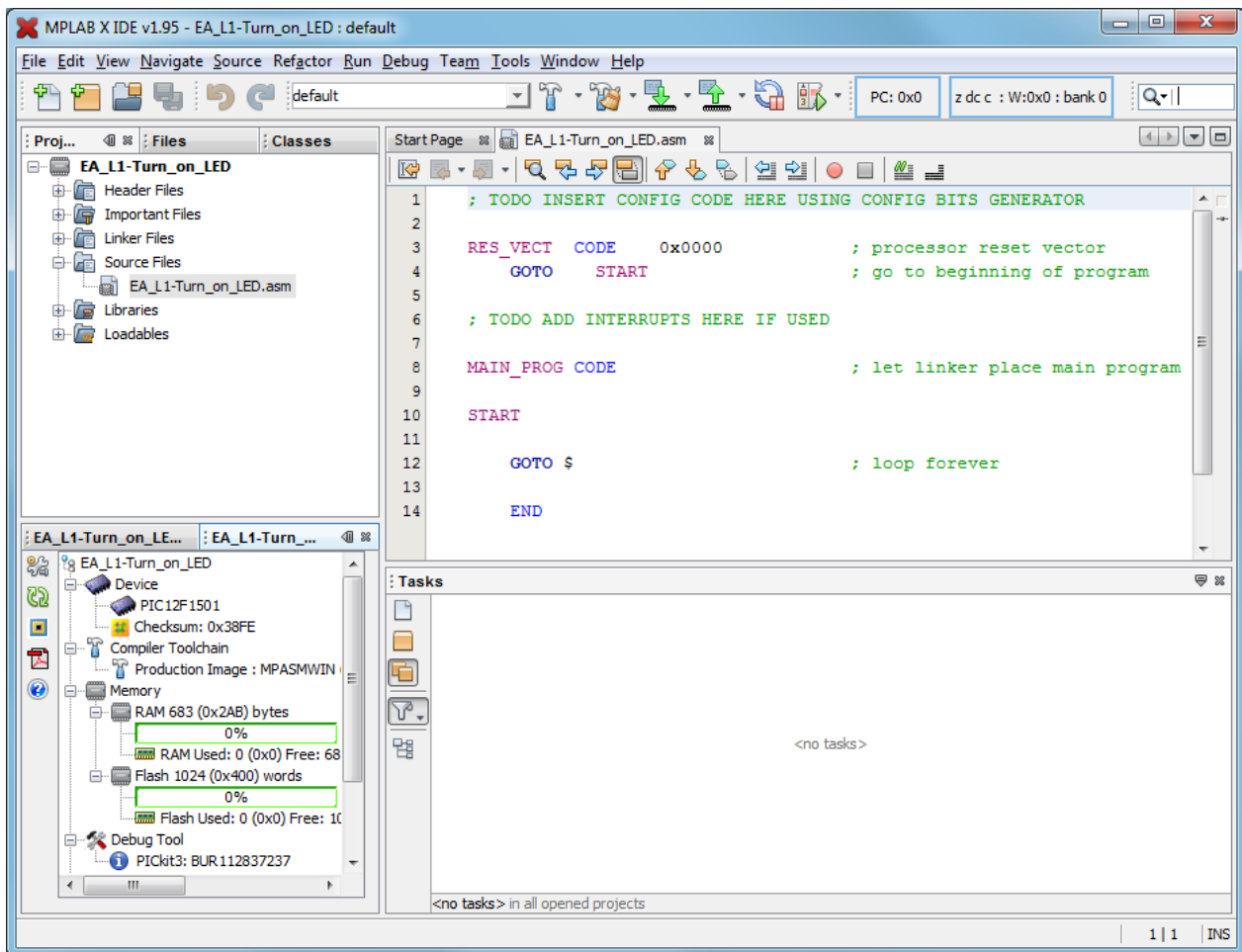


The “Folder” field allows you to place your file in a different directory, not necessarily within the project folder. You wouldn’t normally do that, so it’s ok to leave it blank, as shown here.

The second way to create a new assembler source file is to right-click ‘Source Files’ in the project tree, and select “New → pic_8b_simple.asm...”:



Either way, your source file should now appear under ‘Source Files’ in the project tree, and you should be able to see the source code in the editor window, as shown:



If you don't see the source code, you may need to click on the tab at the top of the editor pane, or double-click the source file name in the project tree.

Now you can finally start working on your code!

Program Code

As we'll see in lesson 3, a large program can consist of a number of source files, each containing various modules or definitions, or, in a simple example such as this one, you may have only a single source file.

Regardless of whether a source file stands on its own or is part of a larger program, it is usual to begin it with a block of comments, providing essential information about the source file such as what it's called, the last modification date and current version (and sometimes a history of previous versions, what has changed in this version, and who changed it), who wrote it, and a general description of what the program or module does.

It can also be useful to include a ‘Files required’ section. This is helpful in larger projects, where your code may rely on other files or modules; you can list any dependencies here.

It is also a good idea to include information on what processor this code is written for; useful if you move it to a different PIC later. You should also document what each pin is used for. It's common, when working on a project, to change the pin assignments – often to simplify the circuit layout. Clearly documenting the pin assignments helps to avoid making mistakes when they are changed!

As mentioned earlier, MPASM comments begin with a ‘;’. They can start anywhere on a line. Anything after a ‘;’ is ignored by the assembler.

The following comment block illustrates the sort of information you should include at the start of each source file:

```

;*****
;
;  Filename:      EA_L1_1-Turn_on_LED.asm
;  Date:         15/9/13
;  File Version: 0.1
;
;  Author:       David Meiklejohn
;  Company:     Gooligum Electronics
;
;*****
;
;  Architecture: Enhanced Mid-range PIC
;  Processor:   12F1501
;
;*****
;
;  Files required: none
;
;*****
;
;  Description:   Lesson 1, example 1
;
;  Turns on LED. LED remains on until power is removed.
;
;*****
;
;  Pin assignments:
;    RA1 = indicator LED
;
;*****

```

Note that the file version is ‘0.1’. I don’t call anything ‘version 1.0’ until it works; when I first start development I use ‘0.1’. You can use whatever scheme makes sense to you, as long as you’re consistent.

You can type these comments into the start of the source code in the editor pane – or, better, copy and paste them from the source file provided with this lesson.

The first line of the “pic_8b_simple.asm” file states:

```
; TODO INSERT CONFIG CODE HERE USING CONFIG BITS GENERATOR
```

This is reminding us that our program should start with processor configuration code.

The 12F1501 has a number of options that are selected by setting various bits in a pair of “configuration words”, sometimes known as “fuses”, which sit outside the normal address space.

The `__CONFIG` assembler directive is used to specify these configuration bits.

We could look up the configuration bits in the data sheet and type in the appropriate `__CONFIG` directives ourselves – and in fact, when you’re creating a new program, based on one that you’ve worked on before (as you’ll often do), it’s quite normal to directly edit the `__CONFIG` directives.

But as this comment suggests, MPLAB X includes a generator which can create these directives for us.

To use it, select the “Window → PIC Memory Views → Configuration Bits” menu item.

You will see the processor configuration options in the “Configuration Bits” window under the editor pane:

Address	Name	Value	Field	Option	Category	Setting
8007	CONFIG1	FFE4	FOSC	INTOSC	Oscillator Selection Bits	INTOSC oscillator: I/O function on CLKIN pin
			WDTE	OFF	Watchdog Timer Enable	WDT disabled
			PWRT	OFF	Power-up Timer Enable	PWRT disabled
			MCLRE	ON	MCLR Pin Function Select	MCLR/VPP pin function is MCLR
			CP	OFF	Flash Program Memory Code Protection	Program memory code protection is disabled
			BOREN	ON	Brown-out Reset Enable	Brown-out Reset enabled
			CLKOUTEN	OFF	Clock Out Enable	CLKOUT function is disabled. I/O or oscillator function on the CLKOUT pin
8008	CONFIG2	DFFF	WRT	OFF	Flash Memory Self-Write Protection	Write protection off
			STVREN	ON	Stack Overflow/Underflow Reset Enable	Stack Overflow or Underflow will cause a Reset
			BORV	LO	Brown-out Reset Voltage Selection	Brown-out Reset Voltage (Vbor), low trip point selected.
			LPBOR	OFF	Low-Power Brown Out Reset	Low-Power BOR is disabled
			LVP	OFF	Low-Voltage Programming Enable	High-voltage on MCLR/VPP must be used for programming

As you can see, there are quite a few options, but you only need to change three (shown in red, above):

FOSC = INTOSC

WDTE = OFF

LVP = OFF

When you have made these selections, click on the ‘Generate Source Code to Output’ button.

The generated source code will appear in the “Config Bits Source” tab in the “Output” window:

```

; PIC12F1501 Configuration Bit Settings

#include "p12F1501.inc"

; CONFIG1
; __config 0xFFE4
__CONFIG __CONFIG1, _FOSC_INTOSC & _WDTE_OFF & _PWRT_OFF & _MCLRE_ON & _CP_OFF & _BOREN_ON & _CLKOUTEN_OFF
; CONFIG2
; __config 0xDFFF
__CONFIG __CONFIG2, _WRT_OFF & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF

```

You can then copy and paste (using the usual select and right-click method) this into your source code in the editor window, replacing the “INSERT CONFIG CODE HERE” comment.

The first line of this generated code is:

```
#include "p12F1501.inc"
```

The ‘#include’ directive causes an *include file* (‘p12F1501.inc’, located in the ‘mpasmx’ folder within the MPASM install directory¹⁰) to be read by the assembler. This file sets up aliases for all the features of

¹⁰ If you are using Windows, this will typically be ‘C:\Program Files\Microchip\MPLABX\mpasmx’

the processor, so that we can refer to registers etc. by name (e.g. 'PORTA') instead of numbers. Lesson 6 explains how this is done; for now we'll simply use these pre-defined names, or *labels*.

If the filename specified in the '#include' directive contains spaces, it must be enclosed in quotes (as shown) or in angle brackets (<>).

Next in the generated code is:

```
; CONFIG1
; __config 0xFFE4
__CONFIG __CONFIG1, _FOSC_INTOSC & _WDTE_OFF & _PWRTE_OFF & _MCLRE_ON &
_CP_OFF & _BOREN_ON & _CLKOUTEN_OFF
; CONFIG2
; __config 0xDFFF
__CONFIG __CONFIG2, _WRT_OFF & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF
```

These directives specify the processor configuration.

[note that each __CONFIG directive must be written as a single line in the assembler source code, not split across multiple lines as the first directive here appears to be]

The 12F1501 has too many configuration options to fit into a single 14-bit word, so it has two configuration words, each defined via a __CONFIG directive with a parameter, '_CONFIG1' or '_CONFIG2' specifying which configuration word is being defined, as shown.

We'll examine these in greater detail in later lessons, but briefly the options being set here are:

- `_FOSC_INTOSC`

This selects the internal RC oscillator as the clock source.

Every processor needs a clock – a regular source of cycles, used to trigger processor operations such as fetching the next program instruction.

Most modern PICs, including the 12F1501, include an internal 'RC' oscillator, which can be used as the simplest possible clock source, since it's all on the chip! It's built from passive components – resistors and capacitors – hence the name RC.

The internal RC oscillator on the 12F1501 runs at approximately 16 MHz and by default this is divided down to 500 kHz. Program instructions are processed at one quarter this speed: 125 kHz, or 8 µs per instruction.

Alternatively, the 12F1501 can use a (possibly more accurate) external clock signal, via the CLKIN pin. This shares its physical pin with RA5, so if you're using an external clock, you can't use the RA5 pin for I/O.

To turn on an LED, we don't need accurate timing, so we'll use the internal RC oscillator.

- `_CLKOUTEN_OFF`

Regardless of whether the internal RC oscillator or an external clock signal is used as the processor clock (FOSC) source, the instruction clock (FOSC/4) can optionally be output on the CLKOUT pin, to allow other devices to be synchronised with the PIC's operation. CLKOUT shares its pin with RA4, so if you're using the clock out facility, you can't use RA4 for I/O.

We don't need to use CLKOUT, so we will leave this feature disabled.

- `_MCLRE_ON`

Enables external reset, or "master clear" ($\overline{\text{MCLR}}$) on pin 4.

If external reset is enabled, pulling this pin low will reset the processor. Or, if external reset is disabled, the pin can be used as an input: RA3. That's why, on the circuit diagram, pin 4 is

labelled “RA3/ $\overline{\text{MCLR}}$ ”; it can be either an input pin or an external reset, depending on the setting of this configuration bit.

The Gooligum training board includes a pushbutton which will pull pin 4 low when pressed, resetting the PIC if external reset is enabled. The PICkit 3 is also able to pull the reset line low, allowing MPLAB to control $\overline{\text{MCLR}}$ (if enabled) – useful for starting and stopping your program.

So unless you need to use every pin for I/O, it’s a good idea to enable external reset by including ‘`_MCLRE_ON`’ in the `__CONFIG` directive.

- `_CP_OFF`
Turns off code protection.

When your code is in production and you’re selling PIC-based products, you may not want competitors stealing your code. If you specify `_CP_ON` instead, your code will be protected, meaning that if someone tries to use a PIC programmer to read it, all they will see are zeros.

Since we’re not designing anything for sale, we’ll make our lives easier by leaving code protection turned off.

- `_WDTE_OFF`
Disables the watchdog timer.

This is a way of automatically restarting a crashed program; if the program is running properly, it continually resets the watchdog timer. If the timer is allowed to expire, the program isn’t doing what it should, so the chip is reset and the crashed program restarted – see lesson 8.

The watchdog timer is very useful in production systems, but a nuisance when prototyping, so we’ll leave it disabled.

- `_BOREN_ON`
Enables brown-out resets.

The PIC’s operation can become unreliable if the supply voltage drops too low, which can happen during a *brown-out*, when the supply voltage sags, but does not fall quickly to zero. The 12F1501 has brown-out detect circuitry, which will reset the PIC in a brown-out situation, if `_BOREN_ON` is selected.

Although your power supply is not likely to suffer from brown-outs, it doesn’t hurt to leave this option enabled – just in case.

- `_BORV_LO`
Selects the low brown-out reset voltage option

This option selects the voltage level at which the brown-out reset (if enabled) will be tripped.

- `_LPBOR_OFF`
Disables low-power brown-out resets.

The 12F1501 also has a lower-power brown-out reset facility; we can leave it disabled.

- `_PWRTE_OFF`
Disables the power-up timer.

When a power supply is first turned on, it can take a while for the supply voltage to stabilise, during which time the PIC’s operation may be unreliable. If the power-up timer is enabled, the PIC is held in reset (it does not begin running the user program) for some time, nominally 64 ms, after the supply voltage reaches a minimum level.

However, since we have enabled the brown-out reset facility, which will prevent the device from starting until the supply voltage is high enough, we don’t need to also enable the power-up timer.

- `_WRT_OFF`
Disables flash memory write protection.
Many newer PIC devices, including the 12F1501, are capable of writing to their flash (program) memory. This is useful in a number of situations, including boot loaders, which allow program firmware to be updated easily in the field, or to store data long term (flash memory being non-volatile).
Of course, you don't want your program to be overwritten by mistake! To prevent that from happening, you may wish to write-protect all or some of the flash memory.
Nevertheless, it's safe in this example to leave flash write protection disabled.
- `_STVREN_ON`
Enables stack overflow/underflow resets.
As we'll see in lesson 3, the *stack* is a special set of registers used when calling subroutines.
Although we won't be using the stack in this example, it doesn't hurt to leave this feature enabled.
- `_LVP_OFF`
Disables low-voltage programming.
Normally, to program the device, a high voltage (around 12 V) must be applied to the VPP pin. Low-voltage programming mode avoids the need for this high voltage, but we don't need it because the PICkit 3 can operate in the traditional high-voltage programming mode.

The comments in the generated configuration code aren't very useful; unless you remember what all of these symbols mean, it's hard to know how the device is being configured.

So we'll add comments and rearrange the configuration code to make it more readable:

```
;***** CONFIGURATION
        ; ext reset, internal oscillator (no clock out), no watchdog,
        ;   brownout resets on, no power-up timer, no code protect
        ; no write protection, stack resets on, low brownout voltage,
        ;   no low-power brownout reset, high-voltage programming
__CONFIG __CONFIG1, _MCLRE_ON & _FOSC_INTOSC & _CLKOUTEN_OFF & _WDTE_OFF &
__BOREN_ON & _PWRTE_OFF & _CP_OFF
__CONFIG __CONFIG2, _WRT_OFF & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF
```

Next, in the "pic_8b_simple.asm" template code, we have:

```
RES_VECT CODE    0x0000          ; processor reset vector
        GOTO     START          ; go to beginning of program
```

The `CODE` directive is used to introduce a *section* of program code.

The `0x0000` after `CODE` is an address in hexadecimal (signified in MPASM by the '0x' prefix).

As mentioned earlier, when the 12F1501 is powered on or reset, it begins the program instruction at address `0x0000`. This is referred to as the *processor reset vector* – it's where the program starts running.

This `CODE` directive is telling the *linker*, which decides where to locate each code section in program memory, to place this section of code at `0x0000`.

In other words, this is the section of code that will run whenever the program starts.

So the first instruction to be executed will be 'GOTO START'.

‘goto’ is an unconditional branch instruction¹¹. It tells the PIC to **go to** a specified program address.

Why begin the program by immediately jumping somewhere else?

There is a clue in the following comment in the template code:

```
; TODO ADD INTERRUPTS HERE IF USED
```

An *interrupt* is a means of interrupting the main program flow in response to an event, such as a change on an input pin, can be responded to immediately.

As we’ll see in lesson 7, when the interrupt is triggered, program execution jumps to an *interrupt service routine (ISR)*, which, in the enhanced mid-range PIC architecture, is always located at address 0004h.

This means that the ISR, or at least its entry point, must be located at address 0004h. But our main program code starts at address 0000h. Unless the main code is only four bytes long, it’s going to run into the address where we have to place the ISR – unless the main program is located somewhere else in memory, where it won’t conflict with the ISR, in which case, when the PIC starts running, the first thing to do is to jump to where the main program is. Hence the ‘GOTO START’ instruction.

We won’t be using interrupts in this example, so we can remove this comment about interrupts, and also remove the ‘GOTO START’, because, with no interrupts, there is no ISR to jump around.

The next piece of the template code represents the start of the main program code:

```
MAIN_PROG CODE                ; let linker place main program
START
```

Using the `CODE` directive like this, without specifying a section address (such as 0x0000), allows the linker to place this section wherever it best fits into program memory.

That’s fine, but since we’re not using interrupts and therefore don’t have an ISR sitting at 0x0004, our main program code might as well continue on directly from the reset vector at 0x0000 – there is no need to create a separate code segment to hold the main program.

So we can remove this `CODE` directive.

It doesn’t hurt to keep the ‘START’ label though – it helps to show where the program starts, and, being no more than a label, doesn’t use any memory.

The template code finishes with a “do nothing” loop:

```
GOTO $                        ; loop forever
END
```

Programs running on small microcontrollers, such as enhanced mid-range PICs, have nowhere to go if they “finish” – it’s not like a PC with an operating system that the program can return control to. If a PIC program is allowed to run past its “end”, it will attempt to execute whichever “instructions” happen to be

¹¹ PIC assembler instructions are not case-sensitive – ‘goto’ and ‘GOTO’ are both acceptable, although lowercase has traditionally be used for instructions, so we’ll use lowercase in these tutorials. Labels, such as ‘START’ may or may not be (depending on the assembler configuration) case sensitive, so it’s best be consistent then using them – if you use the label ‘START’, don’t ever refer to it, or any other label, as ‘start’ or ‘Start’.

in the (uninitialised, non-programmed) remainder of the program memory. Whatever the “program” is doing at that point, it’s not under your control and not behaviour that you want.

So, programs running on small microcontrollers, where there is no operating system, are almost always designed to never finish running. Instead, the usual structure is to have some initialisation code which runs when the program starts, often some interrupt services routines which will be run when interrupts are triggered, and a *main loop*, which repeats the same processes “forever” – that is, until the power is cut off or the device is reset.

This fragment of code isn’t really a “main loop”. Instead we have a one-line “endless” or “infinite” loop, an instruction that does nothing other than continually looping back onto itself, to stop the program from running past this point into uninitialised program memory.

Such a loop could be written as:

```
here    goto    here
```

‘here’ is a label representing the address of the `goto` instruction.

A shorthand way of writing the same thing, that doesn’t need a unique label, is:

```
        goto    $          ; loop forever
```

‘\$’ is an assembler symbol meaning the current program address.

So this line will endlessly loop back on itself.

‘END’ is an assembler directive, marking the end of the program source. The assembler will ignore any text after the ‘END’ directive – so it really should go right at the end of the program!

Of course, we need to replace these example instructions with our own. This is where we place the code to turn on the LED!

Turning on the LED

To turn on the LED on RA1, we need to do two things:

- Configure RA1 as an output
- Set RA1 to output a high voltage

We could leave the other pins configured as inputs, or set them to output a low. Since, in this circuit, they are not connected to anything, it doesn’t really matter. But for the sake of this exercise, we’ll configure them as inputs.

When an enhanced mid-range PIC is powered on, all pins are configured by default as inputs, and the content of the port register, `PORTA`, is undefined.

To configure only RA1 as an output, we have to clear bit 1 of the `TRISA` register, leaving all the other bits in `TRISA` set. This is done by:

```
        movlw   b'111101'      ; configure RA1 (only) as an output
        banksel TRISA
        movwf   TRISA
```

Note again that to specify a binary number in MPASM, the syntax `b‘binary digits’` is used, as shown.

To make RA1 output a ‘high’, we have to set bit 1 of PORTA to ‘1’, which we could do with:

```
movlw    b'000010'      ; set RA1 high
banksel  PORTA
movwf    PORTA
```

Although there is no risk of running into read-modify-write problems when updating the port register in a single write operation like this, to avoid potential problems in other situations it is better to get into the habit of only ever writing to LATA to modify output pins:

```
movlw    b'000010'      ; set RA1 high
banksel  LATA
movwf    LATA
```

Finally, as explained earlier, if we leave it there, when the program gets to the end of this code, it will attempt to execute whatever happens to be in the remainder of the program memory. We need to get the PIC to just sit doing nothing, indefinitely, with the LED still turned on, until it is powered off – which means finishing with an endless loop, as above:

```
goto     $              ; loop forever
```

And of course an ‘END’ directive has to go at the end of the source code.

Once again, this little program has a structure common to most PIC programs: an initialisation section, where the I/O pins and other facilities are configured and initialised, followed by a “main loop”, which repeats forever. Although we’ll add to it in future lessons, we’ll always keep this basic structure of initialisation code followed by a main loop.

Complete program

Putting together all the above, and adding a few more comments, here’s the complete assembler source for turning on an LED, for the PIC12F1501:

```
*****
; Description:      Lesson 1, example 1
;
; Turns on LED.   LED remains on until power is removed.
;
*****
;
; Pin assignments:
;   RA1 = indicator LED
;
*****

#include "p12F1501.inc"

***** CONFIGURATION
; ext reset, internal oscillator (no clock out), no watchdog,
; brownout resets on, no power-up timer, no code protect
; no write protection, stack resets on, low brownout voltage,
; no low-power brownout reset, high-voltage programming
__CONFIG __CONFIG1, _MCLRE_ON & _FOSC_INTOSC & _CLKOUTEN_OFF & _WDTE_OFF &
_BOREN_ON & _PWRTE_OFF & _CP_OFF
__CONFIG __CONFIG2, _WRT_OFF & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF
```

```

;***** RESET VECTOR *****
RES_VECT CODE    0x0000          ; processor reset vector

;***** MAIN PROGRAM *****

;***** Initialisation
start
    ; configure port
    movlw  b'111101'          ; configure RA1 (only) as an output
    banksel TRISA
    movwf  TRISA

;***** Main code
    ; turn on LED
    movlw  b'000010'          ; set RA1 high
    banksel LATA
    movwf  LATA

    ; loop forever
    goto  $

    END

```

That's it! Not a lot of code, really...

Building the Application and Programming the PIC

Now that we have the complete assembler source, we can build the final application code and program it into the PIC.

This is done in two steps:

- Build the project
- Use a programmer to load the program code into the PIC

The first step, building the project, involves assembling the source files¹² to create object files, and linking these object files, to build the executable code. Normally this is transparent; MPLAB does all of this for you in a single operation. The fact that, behind the scenes, there are multiple steps only becomes important when you start working with projects that consist of multiple source files or libraries of pre-assembled routines.

A PIC programmer, such as the PICkit 3, is then used to upload the executable code into the PIC. Although a separate application is sometimes used for this “programming” process, it's convenient when developing code to do the programming step from within MPLAB, which is what we'll look at here.

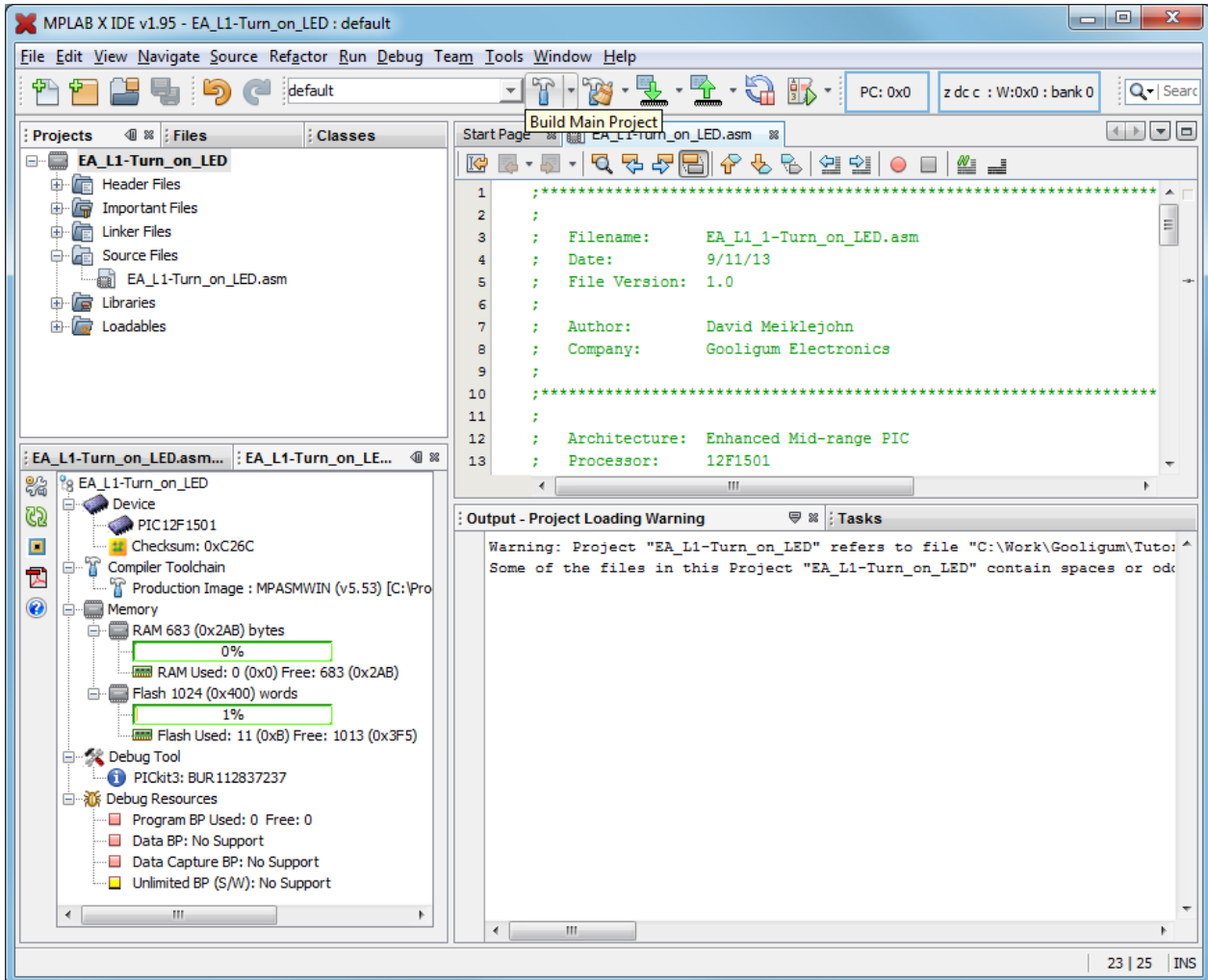
Building the project

Before you build your project using MPLAB X, you should first ensure that it is the “main” project. It should be highlighted in bold in the Projects window.

¹² Although there is only one source file in this simple example, larger projects often consist of multiple files, as mentioned earlier; we'll see an example in lesson 3.

To set the project you want to work on (and build) as the main project, you should right-click it and select “Set as Main Project”. If you happen to have more than one project in your project window, you can by removing any project you are not actively working on (to reduce the chance of confusion) from the Projects window, by right-clicking it and selecting “Close”.

To build the project, right-click it in the Projects window and select “Build”, or select the “Run → Build Main Project” menu item, or simply click on the “Build Main Project” button (looks like a hammer) in the toolbar:



This will assemble any source files which have changed since the project was last built, and link them.

An alternative is “Clean and Build”, which removes any assembled (object) files and then re-assembles all files, regardless of whether they have been changed. This action is available by right-clicking the project in the Projects window, or under the “Run” menu, or by clicking on the “Clean and Build Main Project” button (looks like a hammer with a brush) in the toolbar.

When you build the project, you’ll see messages in the Output window, showing your source files being assembled and linked. Toward the end, you should see:

BUILD SUCCESSFUL (total time: 2s)

(of course, your total time will probably be different...)

If, instead, you see an error message, you’ll need to check your code and your project configuration.

You are, however, likely to see some warning messages, similar to:

```
Message[302] C:\...\EA_L1-TURN_ON_LED.ASM 55 : Register in operand not in bank
0. Ensure that bank bits are correct.
Message[303] C:\...\EA_L1-TURN_ON_LED.ASM 67 : Program word too large.
Truncated to core size. (FFE4)
```

The [302] messages are generated whenever your code references a register which is not in bank 0, to remind you that you should be taking care to set the bank selection bits correctly. Since we have been using the `banksel` directive to ensure that the correct bank is selected, it can be annoying to see these messages – particularly in a larger program, where there will be many more of them. And worse, having a large number of unnecessary messages can make it easy to miss more important messages and warnings.

Luckily, messages and warnings can be disabled, using the ‘`errorlevel`’ directive:

```
errorlevel -302 ; no warnings about registers not in bank 0
```

The [303] messages are generated because the processor configuration constants are defined in the ‘`p12f1501.inc`’ include file as 16-bit values, while the 12F1501’s configuration words are only 14 bits wide. 16 bit into 14 don’t go, so the assembler has to *truncate* the config values to 14 bits by dropping the top two bits.

We don’t need to see these warnings either, so it’s ok to disable them also:

```
errorlevel -303 ; no warnings about program word too large
```

These directives should be placed toward the beginning of your program, before the `__CONFIG` directives, as follows:

```
#include "p12F1501.inc"

errorlevel -302 ; no warnings about registers not in bank 0
errorlevel -303 ; no warnings about program word too large

;***** CONFIGURATION
; ext reset, internal oscillator (no clock out), no watchdog,
; brownout resets on, no power-up timer, no code protect
; no write protection, stack resets on, low brownout voltage,
; no low-power brownout reset, high-voltage programming
__CONFIG __CONFIG1, _MCLRE_ON & _FOSC_INTOSC & _CLKOUTEN_OFF & _WDTE_OFF &
_BOREN_ON & _PWRTE_OFF & _CP_OFF
__CONFIG __CONFIG2, _WRT_OFF & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF
```

If you now build the project again, you should no longer see any [302] or [303] messages.

Programming the PIC

The final step is to upload the executable code into the PIC.

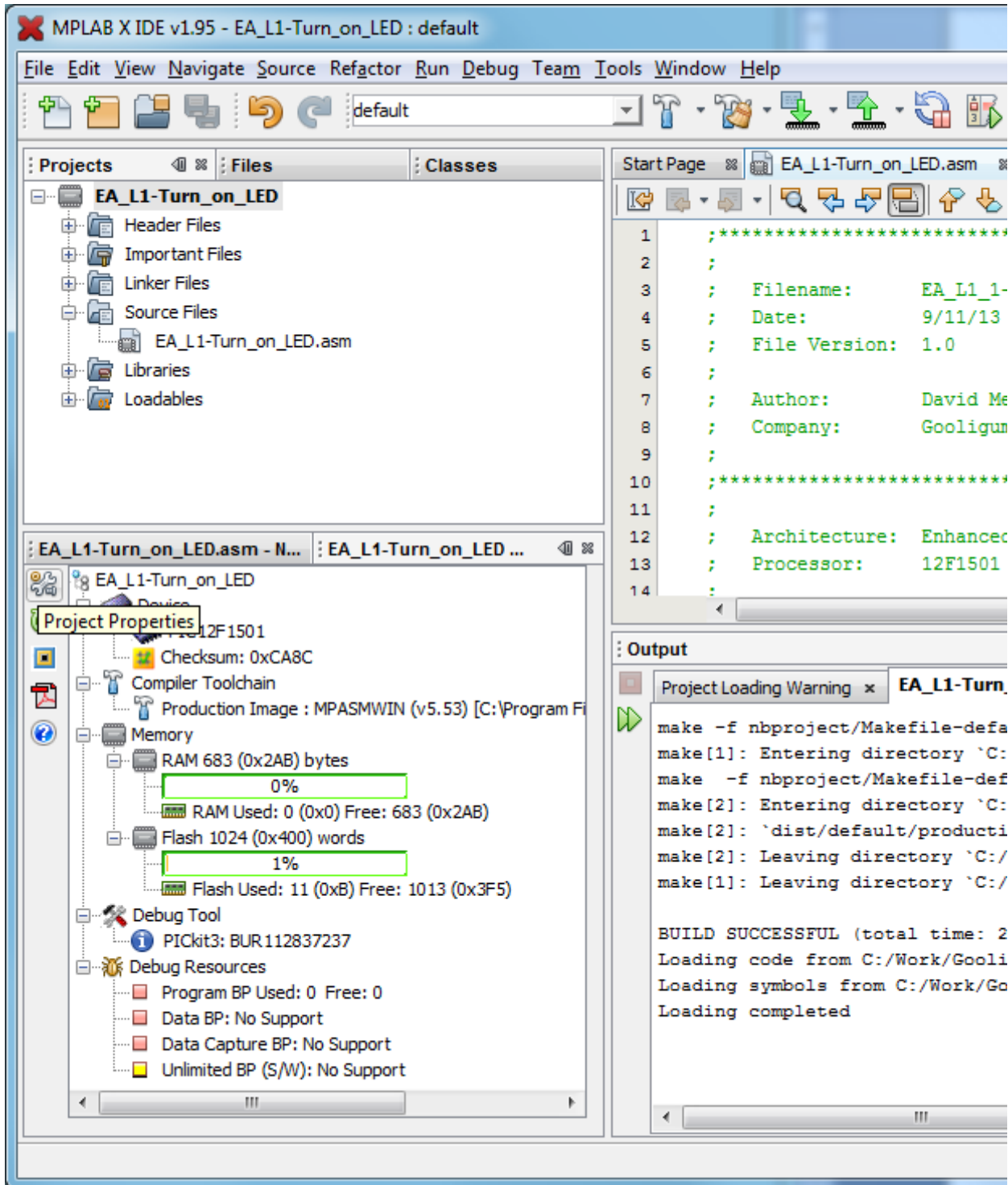
First, ensure that you have connected your PICkit 3 programmer to your Gooligum training board or Microchip LPC Demo Board, with the PIC correctly installed in the appropriate IC socket¹³, and that the programmer is plugged into your PC.

¹³ Or, in general, that the PIC you wish to program is connected to whichever programmer or debugger you are using, whether it’s in a demo/development/training board, a production board, or a standalone programmer.

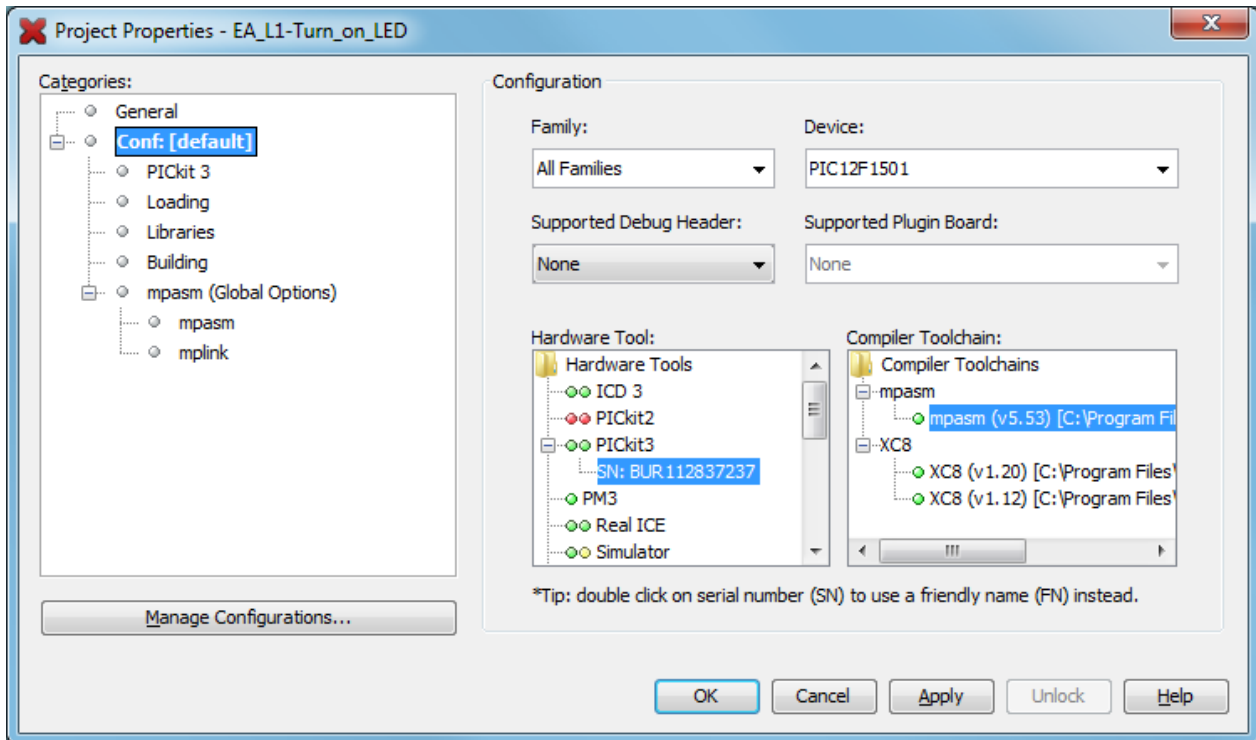
If you have been following this lesson, you will have specified the programmer when you created your project (in step 4 of the wizard).

The project dashboard, in the panel in the bottom right of the workspace, shows the currently-selected programmer under “Debug Tool”.

If you want to change this tool selection, you can right-click your project in the Projects window and select “Properties”, or simply click on the “Project Properties” button on the left side of the project dashboard, as shown:



This will open the project properties window, where you can verify or change your hardware tool (programmer) selection:



After closing the project properties window, you can now program the PIC.

You can do this by right-clicking your project in the Projects window, and select “Make and Program Device”. This will repeat the project build, which we did earlier, but because nothing has changed (we have not edited the code), the “make” command will decide that there is nothing to do, and the assembler will not run.

Instead, in the “Build, Load” tab in the Output pane you should see output like:

```
BUILD SUCCESSFUL (total time: 1s)
Loading code from C:/Work/Gooligum/Tutorials/Series 2/Web/Enhanced/1 - Light an LED/EA_L1-Turn_on_LED.X/dist/default/production/EA_L1-Turn_on_LED.X.production.hex...
Loading symbols from C:/Work/Gooligum/Tutorials/Series 2/Web/Enhanced/1 - Light an LED/EA_L1-Turn_on_LED.X/dist/default/production/EA_L1-Turn_on_LED.X.production.cof...
Loading completed
Connecting to programmer...
Programming target...
```

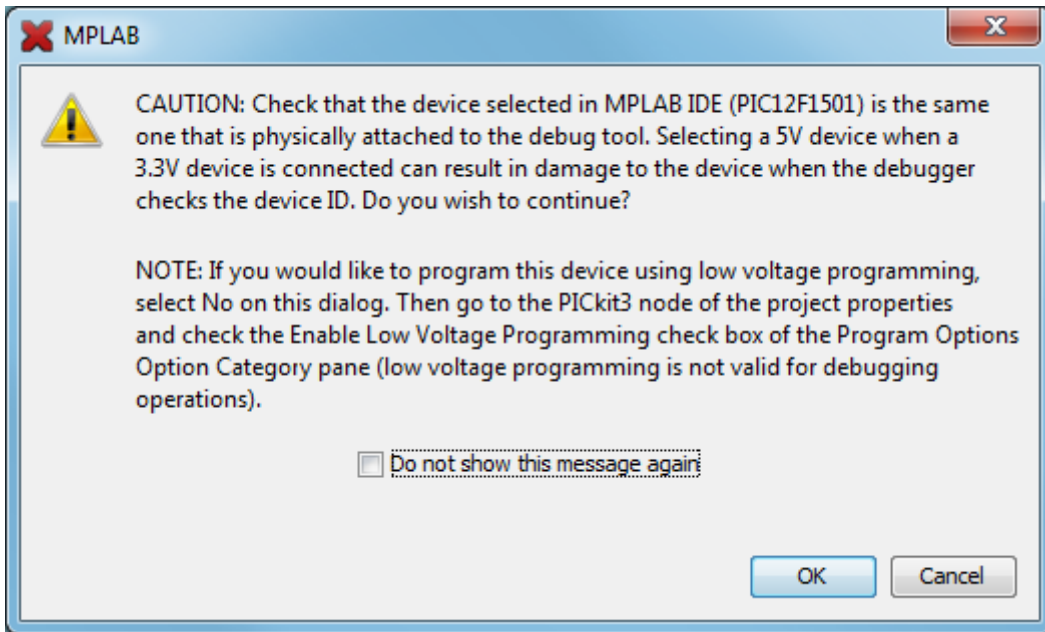
(the total time is smaller than before, because no assembly had to be done).

A “PICKit 3” tab will also appear in the Output pane, where you can see what the PICKit 3 is doing.

Your PICKit 3 may need to have new firmware downloaded into it, to allow it to program enhanced mid-range devices, in which case you will see messages like:

```
Downloading Firmware...
Downloading RS...
Downloading AP...
AP download complete
Programming download...
Firmware Suite Version.....01.29.33
Firmware type.....Enhanced Midrange
```

You may also see a voltage caution warning, as shown below:



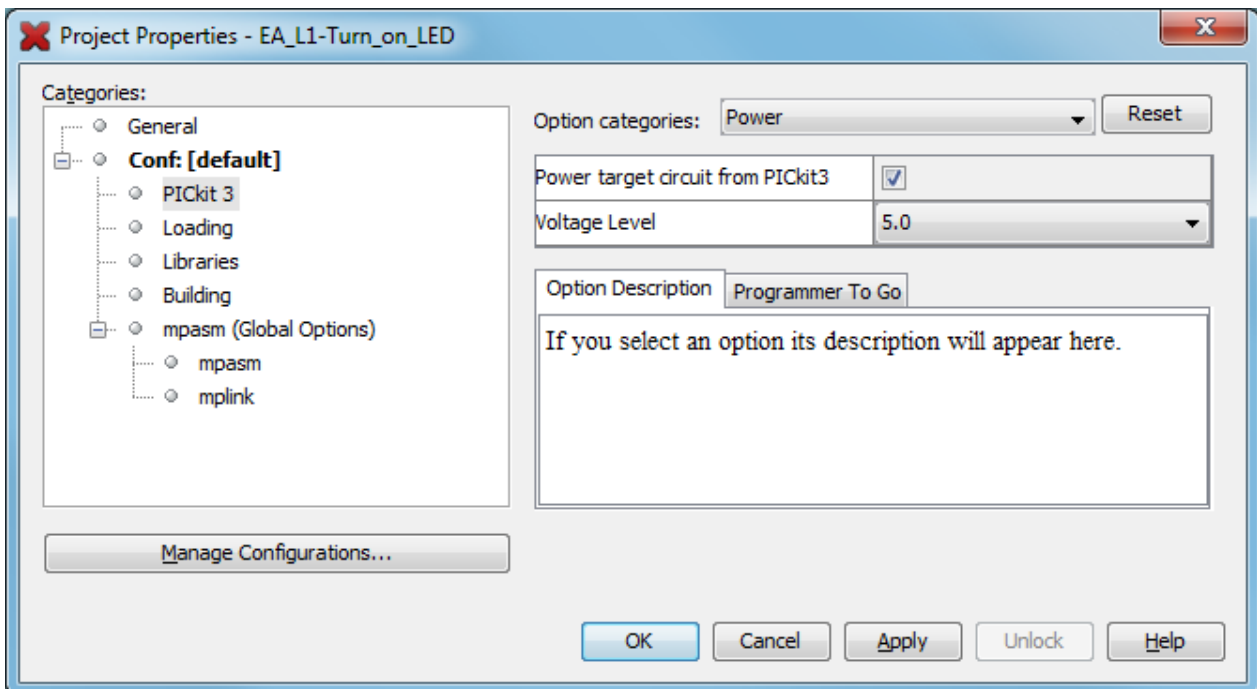
Since we are using a 5 V device, you can click ‘OK’. And feel free to click “Do not show this message again”, to avoid seeing this caution every time you program your PIC.

You may now see an error message in the PICKit 3 output tab, stating:

Target device was not found. You must connect to a target device to use PICKit 3.

This happens if the PIC is unpowered, so we need to tell the PICKit 3 to supply power.

Open the project properties window (as on the previous page), select ‘PICKit 3’ in the categories tree, and choose ‘Power’ option in the drop-down option categories list:



Select “Power target circuit from PICKit3”, as shown. You can leave the voltage set to 5.0 V, and then click ‘OK’.

If you now perform “Make and Program Device” again, the programming should be successful and you should see, in the build output tab, messages ending in:

Programming completed


Note that this action combines making (or building) the project, with programming the PIC.

In fact, there is no straightforward way with MPLAB X to simply program the PIC without building your project as well.

This makes sense, because you will almost always want to program your PIC with the latest code. If you make a change in the editor, you want to program that change into the PIC. With MPLAB X, you can be sure that whatever code you see in your editor window is what will be programmed into the PIC.

But most times, you’ll want to go a step further, and run your program, after uploading it into the PIC, to see if it works. For that reason, MPLAB X makes it very easy to build your code, program it into your PIC, and then run it, all in a single operation.

There are a few ways to do this:


- Right-click your project in the Projects window, and select “Run”, or
- Select the “Run → Run Main Project” menu item, or
- Press ‘F6’, or
- Click on the “Make and Program Device” button in the toolbar: 


Whichever of these you choose, you should see output messages ending in:

Running target...

The LED on RA1 should now light.

Being able to build, program and run in a single step, by simply pressing ‘F6’ or clicking on the “Make and Program Device” button is very useful, but what if you don’t want to automatically run your code, immediately after programming?

If you want to avoid running your code, click on the “Hold in Reset” toolbar button () before programming. You can now program your PIC as above.


Your code won’t run until you click the reset toolbar button again, which now looks like  and is now tagged as “Release from Reset”.

Summary

The sections above, on building your project and programming the PIC, have made using MPLAB X seem much more complicated than it really is.

Certainly, there are a lot of options and ways of doing things, but in practice it’s very simple.

Most of the time, you will be working with a single project, and only one hardware tool, such as a programmer or debugger, which you will have selected when you first ran the New Project wizard.

In that case (and most times, it will be), just press ‘F6’ or click on  to build, program and run your code – all in a single, easy step.

That’s all there is to it. Use the New Project wizard to create your project, add a template file to base your code on, use the editor to edit your code, and then press ‘F6’.

Conclusion

For such a simple task as lighting an LED, this has been a very long lesson!

In summary, we:

- Provided an overview of the enhanced mid-range PIC architecture
- Introduced the PIC12F1501
- Showed how to configure and use the PIC's output pins
- Implemented an example circuit using two development boards:
 - Gooligum training and development board
 - Microchip Low Pin Count Demo Board
- Looked at Microchip's assembly template code and saw:
 - some PIC assembler directives
 - some PIC configuration options
 - our first couple of PIC instructions
- Modified it to create our (very simple!) PIC program
- Introduced the MPLAB X integrated development environment
- Showed how to use MPLAB X to:
 - Create a new project, based on a template
 - Modify that template code
 - Build the program
 - Program the PIC, using a PICKit 3
 - Run the program

That does seem to be a lot of theory, to accomplish so little.

Nevertheless, after all this, you have a solid base to build on. You have a working development environment. You can create projects, modify your code, load (program) your code into your PIC, and make it run.

Congratulations! You've taken your first step in PIC development!

That first step is the hardest. From this point, we build on what's come before.

In the [next lesson](#), we'll make the LED flash...

Introduction to PIC Programming

Enhanced Mid-Range Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 2: Flash an LED

In [lesson 1](#) we lit a single LED connected to one of the pins of a PIC12F1501.

Now we'll make it flash.

In doing this, we will learn about:

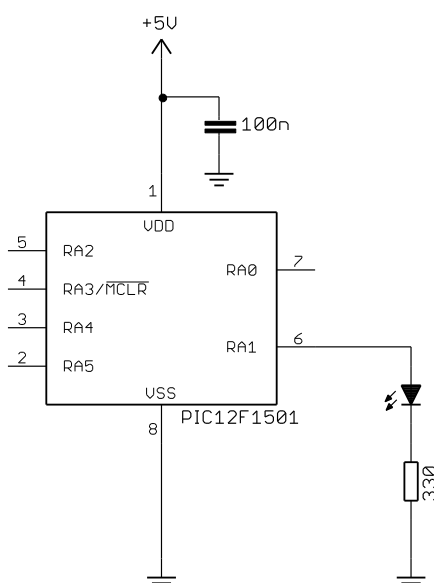
- Selecting the internal oscillator frequency
- Using loops to create delays
- Variables
- Using exclusive-or (xor) to flip bits

The development environments and microcontrollers used for this lesson are the same as those in lesson 1.

Again, it is assumed that you are using a Microchip PICkit 3 programmer and either the [Gooligum Baseline and Mid-range PIC Training and Development Board](#) or Microchip's Low Pin Count (LPC) Demo Board, with Microchip's MPLAB X integrated development environment. But it is of course possible to adapt these instructions to a different programmers and/or development boards.

Example Circuit

Here's the circuit from [lesson 1](#) again:



If you have the Gooligum training board, simply plug the PIC12F1501 into the top section of the 14-pin IC socket – the section marked '12F'.

Connect a shunt across the jumper (JP12) on the LED labelled 'RA1', and ensure that every other jumper is disconnected.

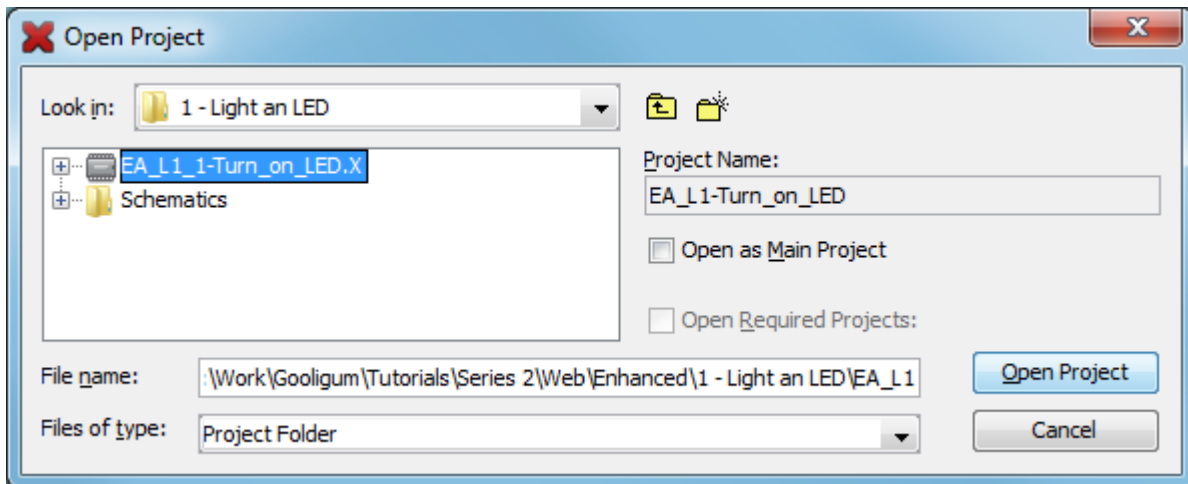
If you are using Microchip's Low Pin Count Demo Board, refer back to [lesson 1](#) to see how to build this circuit, by soldering a resistor, LED (and optional isolating jumper) to the demo board, or by making connections on the demo board's 14-pin header.

Creating a new project in MPLAB X

It is a good idea, where practical, to base a new software project on work you've done before. In this case, it makes sense to build on the program from [lesson 1](#) – we just have to add extra instructions to flash the LED.

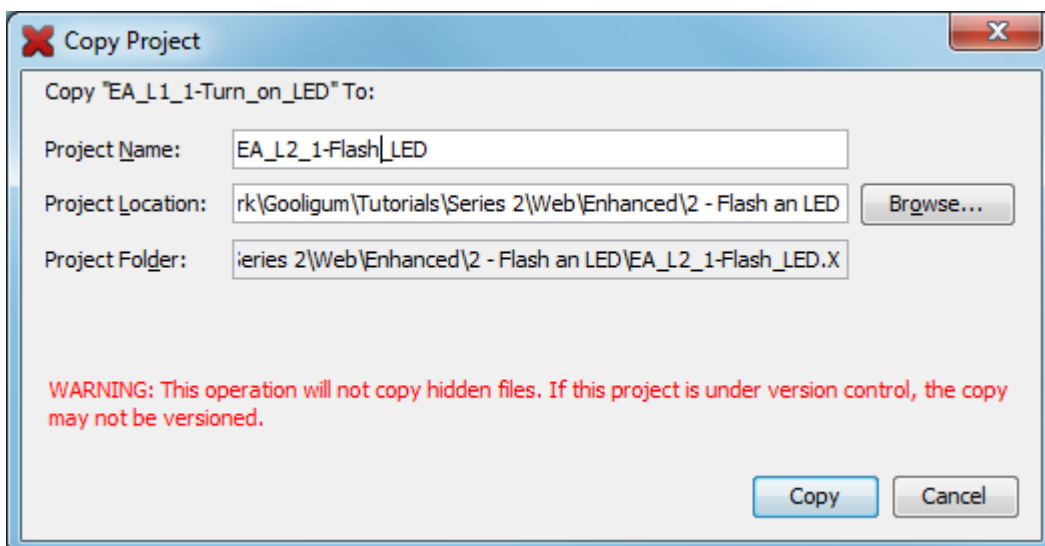
To create a new project in MPLAB X, based on an existing project, you first need to go into MPLAB X and open your existing project.

If you were recently working on the project you want to copy (such as the project from [lesson 1](#)), it is probably already visible in the Projects window. If it's not, it may appear under the “File → Open Recent Project” menu list. Or you can use the “File → Open Project” menu item, or click on the “Open Project...” toolbar button and browse to your project folder, select it, and click ‘Open Project’:



You should now right-click the project name ('EA_L1_1-Turn_on_LED' in this example) in the Projects window, and select “Copy...”.

The “Copy Project” dialog then gives you a chance to give your copied project a new name, such as ‘EA_L2_1-Flash_LED’. You can also specify (and create, if you wish) a new folder for the project location, by browsing to it:

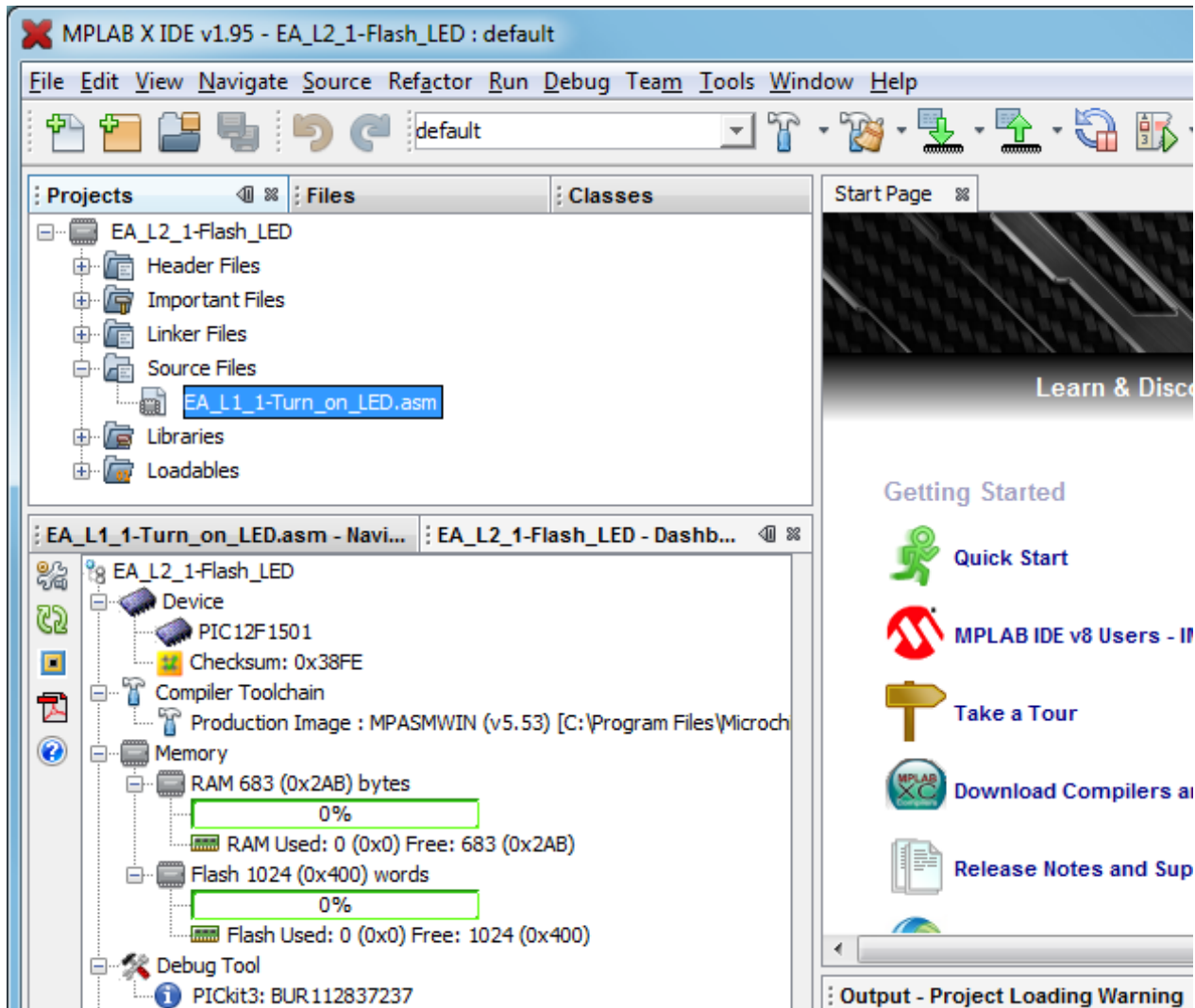


When you are satisfied with your new project name and location, click ‘Copy’.

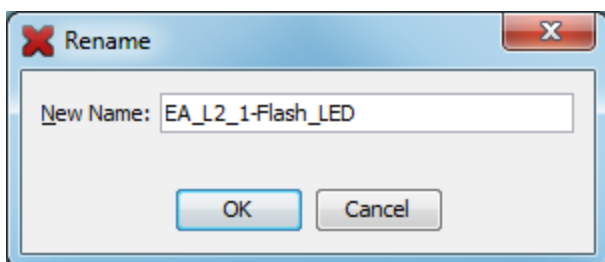
Your new project should now appear in the Projects window.

You can close your old project by right-clicking it and selecting “Close”, so that only your new project is visible.

If you expand your new project, you'll see that source file from the old project has been copied into the new project, with its original name:



To rename the source file, to something more appropriate for this project, right-click it and select “Rename...”:

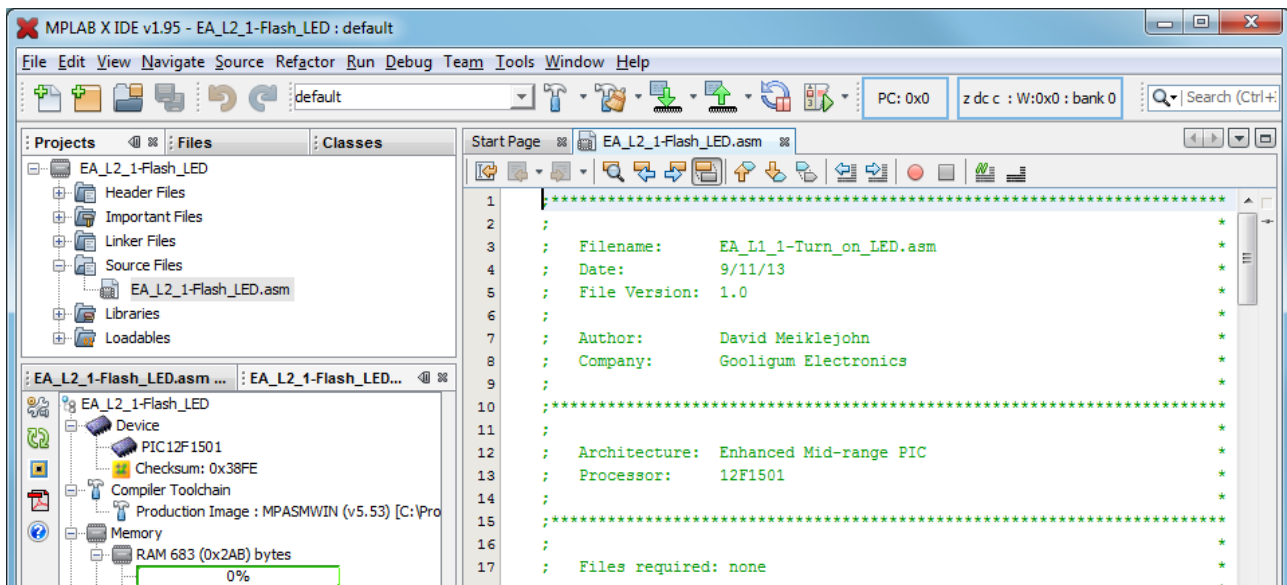


Type in the new name, such as ‘EA_L2_1-Flash_LED’ and then click ‘OK’.

Note that there is no need to type the ‘.ASM’ suffix – the Rename dialog will keep the existing file extension.

You now have a new project, with a new name in a new location, with a renamed source file, copied from your old project.

If you double-click your new source file, you'll see a copy of your code from [lesson 1](#) in an editor window:



Flashing the LED

You can now use the editor to update your code from [lesson 1](#).

We'll need to add some code to make the LED flash, but first the comments should be updated to reflect the new project. For example:

```
;*****
;
;  Filename:      EA_L2_1-Flash_LED.asm
;  Date:         17/11/13
;  File Version:  0.1
;
;  Author:       David Meiklejohn
;  Company:      Gooligum Electronics
;
;*****
;
;  Architecture: Enhanced Mid-range PIC
;  Processor:    12F1501
;
;*****
;
;  Files required: none
;
;*****
;
;  Description:   Lesson 2, example 1
;
;  Flashes an LED at approx 1 Hz.
;  LED continues to flash until power is removed.
;
;*****
;
;  Pin assignments:
;  RA1 = flashing LED
;
;*****
```

We're using the same PIC device as before, and it will be configured the same way, so we can leave the processor include and configuration sections unchanged. There is also no need to change the reset vector section.

So we still have, unchanged from [lesson 1](#):

```
#include "p12F1501.inc"

    errorlevel    -302            ; no warnings about registers not in bank 0
    errorlevel    -303            ; no warnings about program word too large

;***** CONFIGURATION
        ; ext reset, internal oscillator (no clock out), no watchdog,
        ;   brownout resets on, no power-up timer, no code protect
        ; no write protection, stack resets on, low brownout voltage,
        ;   no low-power brownout reset, high-voltage programming
    __CONFIG __CONFIG1, _MCLRE_ON & _FOSC_INTOSC & _CLKOUTEN_OFF & _WDTE_OFF &
    _BOREN_ON & _PWRTE_OFF & _CP_OFF
    __CONFIG __CONFIG2, _WRT_OFF & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF

;***** RESET VECTOR *****
RES_VECT CODE    0x0000            ; processor reset vector
```

Again, we need to set up the PIC so that only RA1 is configured as an output, so we can leave the initialisation code from [lesson 1](#) intact:

```
;***** MAIN PROGRAM *****

;***** Initialisation
start
    ; configure port
    movlw    b'111101'            ; configure RA1 (only) as an output
    banksel TRISA
    movwf    TRISA
```

In [lesson 1](#), we made RA1 high, and left it that way. To make it flash, we need to set it high, then low, and then repeat.

You may think that you could achieve this with something like:

```
flash
    banksel LATA
    movlw    b'000010'            ; set RA1 high
    movwf    LATA
    movlw    b'000000'            ; set RA1 low
    movwf    LATA
    goto     flash                ; repeat forever
```

If you try this code, you'll find that the LED appears to remain on continuously. In fact, it's flashing too fast for the eye to see.

The PIC has been configured to use its internal RC oscillator, which by default provides a 500 kHz processor clock. Each instruction executes in four processor clock cycles, which, given a 500 kHz processor clock, is 8 μ s – except instructions which branch to another location, such as 'goto', which require two instruction cycles, or 16 μ s.

Thus, by default, this loop takes a total of 56 μs , so the LED flashes at $1/(56 \mu\text{s}) = 17.9 \text{ kHz}$. That's much too fast to see!

Again, that's at the default processor clock speed of 500 kHz.

However, the PIC12F1501's internal RC oscillators can be configured, via the **OSCCON** register, to provide a range of processor clock frequencies:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OSCCON	–	IRCF3	IRCF2	IRCF1	IRCF0	–	SCS1	SCS0

The **IRCF** bits are used to select the internal oscillator frequency, as follows:

IRCF<3:0>	Oscillator	Frequency
000x	LF	31 kHz (approx)
001x	HF	31.25 kHz
0100	HF	62.5 kHz
0101	HF	125 kHz
0110	HF	250 kHz
0111	HF	500 kHz (default)
1011	HF	1 MHz
1100	HF	2 MHz
1101	HF	4 MHz
1110	HF	8 MHz
1111	HF	16 MHz

The 12F1501 actually has two internal RC oscillators: an uncalibrated low frequency oscillator, 'LFINTOSC', running at approximately 31 kHz, and a high frequency oscillator, 'HFINTOSC', which is factory-calibrated to run at 16 MHz.

This 16 MHz oscillator is used as the clock source in the remaining "HF" modes, divided by a postscaler to generate frequencies down as low as 31.25 kHz, as shown in the table on the left¹.

The internal clock source (LFINTOSC or HFINTOSC, as above) is selected whenever the **SCS1** bit is set, regardless of the processor configuration words.

Otherwise, if **SCS<1:0> = 00**, the clock source is selected by the oscillator selection bits in the configuration words.

However, even if we select the lowest possible calibrated processor frequency of 31.25 kHz, for a processor cycle time of 32 μs (giving an execution time of $4 \times 32 \mu\text{s} = 128 \mu\text{s}$ for most instructions), the loop will complete in 896 μs , and the LED will flash at 1.17 kHz – still too fast to see.

So to slow it down to a more sedate (and visible!) 1 Hz, we have to add a delay.

But before looking at delays, we can make a small improvement to the code.

To flip, or toggle, a single bit – to change it from 0 to 1 or from 1 to 0, you can exclusive-or it with 1.

That is:

$$0 \text{ XOR } 1 = 1$$

$$1 \text{ XOR } 1 = 0$$

¹ Not all possible **IRCF** values are shown here; those omitted duplicate some of the available processor frequencies.

So to repeatedly toggle RA1, we can read the current state of LATA, exclusive-or the bit corresponding to RA1, then write it back to LATA, as follows:

```

        movlw    b'000010'        ; bit mask to toggle RA1 only
        banksel LATA
flash   xorwf   LATA,f            ; toggle RA1 using mask in W
        goto    flash            ; repeat forever

```

The `xorwf` instruction exclusive-ors the *W* register with the specified register – “exclusive-or *W* with file register”, and writes the result either to the specified file register (LATA in this case) or to *W*.

Many of the PIC instructions, like `xorwf`, are able to place the result of an operation (e.g. add, subtract, or in this case XOR) into either a file register or *W*. This is referred to as the instruction destination. A ‘, *f*’ at the end indicates that the result should be written back to the file register; to place the result in *W*, use ‘, *w*’ instead.

This single instruction – `xorwf LATA,f` – is doing a lot of work. It reads LATA, performs the XOR operation, and then writes the result back to LATA.

Note that in this example there is no need to set RA1 to an initial state; whether it’s high or low to start with, it will be successively flipped. But usually you will want to ensure that the output pins are in a known state before the main loop begins.

For example, if we wanted to begin with the LED off, we would clear the bit in LATA corresponding to RA1. We don’t have any other output pins, but it doesn’t hurt to clear the whole of LATA, in case the LED is moved to another pin or other LEDs added later.

In that case you would include in your initialisation code something like:

```

        banksel LATA
        movlw   b'000000'        ; start with all output pins low
        movwf  LATA              ; (LED off)

```

Or alternatively:

```

        banksel LATA            ; start with all output pins low
        clrf   LATA             ; (LED off)

```

The `clrf` instruction clears (to 0) the specified register – “clear file register”.

It’s usually best to initialise the output pins before they are configured as outputs, so that they do not, even for an instant, output an incorrect level when the program starts running.

So our initialisation code becomes:

```

;***** Initialisation
start
    ; configure port
    banksel LATA            ; start with all output pins low
    clrf   LATA             ; (LED off)
    movlw  b'111101'        ; configure RA1 (only) as an output
    banksel TRISA
    movwf  TRISA

```

Delay Loops

To make the flashing visible, we need to slow it down, and that means getting the PIC to “do nothing” between LED changes.

The enhanced mid-range PICs do have a “do nothing” instruction: ‘nop’ – “no operation”. All it does is to take some time to execute.

As explained above, how much time depends on the clock rate. Instructions are executed at one quarter the rate of the processor clock. If the processor clock is running at 500 kHz, the instructions will be clocked at ¼ of this rate: 125 kHz. Each instruction cycle is then 8 µs.

Most enhanced mid-range PIC instructions, including ‘nop’, execute in a single instruction cycle. The exceptions are those which jump to another location (such as ‘goto’) or if an instruction is conditionally skipped (we’ll see an example of this soon). So, at the default clock speed, a single ‘nop’ provides an 8 µs delay – not very long!

Another “do nothing” instruction is ‘goto \$+1’. Since ‘\$’ stands for the current address, ‘\$+1’ is the address of the next instruction. Hence, ‘goto \$+1’ jumps to the following instruction – apparently useless behaviour. But all ‘goto’ instructions executes in two instruction cycles. So ‘goto \$+1’ is equivalent to two ‘nop’s, but using less program memory.

To flash at 1 Hz, the PIC should light the LED, wait for 0.5 s, turn off the LED, wait for another 0.5 s, and then repeat.

Our code changes the state of the LED once each time around the loop, so we need to add a delay of 0.5 s within the loop. That’s 500,000 µs, or, given a 500 kHz processor clock, 62,500 instruction cycles. Clearly we can’t do that with ‘nop’s or ‘goto’s alone!

The answer, of course, is to use loops to execute instructions enough times to build up a useful delay. But we can’t just use a ‘goto’, or else it would loop forever and the delay would never finish. So we have to loop some finite number of times, and for that we need to be able to count the number of times through the loop (incrementing or decrementing a loop counter variable) and test when the loop is complete.

And that means that we need to define at least one *variable*, to hold the loop counter.

To define variables, we have to allocate (or *reserve*) data memory to store them in.

For example:

```

;***** VARIABLE DEFINITIONS
TEMP_VAR      UDATA
temp1         RES      1           ;example variable definitions
temp2         RES      2

```

The UDATA directive tells the linker that this is the start of a section of uninitialised data. This is data memory space that is simply set aside for use later. The linker will place it somewhere in general purpose RAM. The label, such as ‘TEMP_VAR’ here, is only needed if there is more than one UDATA section.

The RES directive is used to reserve a number of memory locations. Each location in data memory is eight bits, or one byte, wide, so in this case, one byte is being reserved for a variable called ‘temp1’ and two bytes for ‘temp2’. Thus, ‘temp1’ is an 8-bit variable and ‘temp2’ is a 16-bit variable. What type of data, such as an unsigned integer between 0 and 255, or a character, is stored in each variable is up to your program. The variable definitions only allocate space for them.

The address of each variable is assigned when the code is linked (after assembly), and the program can refer to the variable by name (i.e. temp1), without having to know what its address in data memory is.

Here's an example of a simple "do nothing" delay loop:

```

        movlw    .10
        movwf   dc1           ; dc1 = 10 = number of loop iterations
dly1   nop
        decfsz  dc1, f
        goto    dly1

```

The first two instructions write the decimal value "10" to a loop counter variable called 'dc1'.

Note that to specify a decimal value in MPASM, you prefix it with a '.'. If you don't include the '.', the assembler will use the default radix (hexadecimal), and you won't be using the number you think you are! Although it's possible to set the default radix to decimal, you'll run into problems if you rely on a particular default radix and then later copy and paste your code into another project, with a different default radix, giving different results. It's much safer, and clearer, to simply prefix all hexadecimal numbers with '0x' and all decimal numbers with '.'.

The 'decfsz' instruction performs the work of implementing the loop – "decrement file register, skip if zero". First, it decrements the contents of the specified register, writes the result back to the register (as specified by the ', f' destination), then tests whether the result was zero. If it's not yet zero, the next instruction is executed, which will normally be a 'goto' which jumps back to the start of the loop. But if the result of the decrement is zero, the next instruction is skipped; since this is typically a 'goto', skipping it means exiting the loop.

The 'decfsz' instruction normally executes in a single instruction cycle. But if the result is zero, and the next instruction is skipped, an extra cycle is added, making it a two-cycle instruction.

There is also an 'incfsz' instruction, which is equivalent to 'decfsz', except that it increments instead of decrementing. It's used if you want to count up instead of down. For a loop with a fixed number of iterations, counting down is more intuitive than counting up, so 'decfsz' is more commonly used for this.

In the code above, the loop counter, 'dc1', starts at 10. At the end of the first loop, it is decremented to 9, which is non-zero, so the 'goto' instruction is not skipped, and the loop repeats from the 'dly1' label. This process continues – 8,7,6,5,4,3,2 and on the 10th iteration through the loop, dc1 = 1. This time, dc1 is decremented to zero, and the "skip if zero" comes into play. The 'goto' is skipped, and execution continues after the loop.

You can see that the number of loop iterations is equal to the initial value of the loop counter (10 in this example). Call that initial number N. The loop executes N times.

To calculate the total time taken by the loop, add the execution time of each instruction in the loop:

nop		1
decfsz	dc1, f	1 (except when result is zero)
goto	dly1	2

That's a total of 4 cycles, except the last time through the loop, when the decfsz takes an extra cycle and the goto is not executed (saving 2 cycles), meaning the last loop iteration is 1 cycle shorter. And there are two instructions before the loop starts, adding 2 cycles.

Therefore the total delay time = $(N \times 4 - 1 + 2)$ instruction cycles = $(N \times 4 + 1) \times 8 \mu\text{s}^2$.

If there was no 'nop', the delay would be $(N \times 3 + 1)$ cycles; if two 'nop's, then it would be $(N \times 5 + 1)$ cycles, etc.

² assuming a 500 kHz processor clock, for all the time calculations in this section

It may seem that, because 255 is the highest 8-bit number, the maximum number of iterations (N) should be 255. But not quite. If the loop counter is initially 0, then the first time through the loop, the 'decfsz' instruction will decrement it, and if an 8-bit counter is decremented from 0, the result is 255, which is non-zero, and the loop continues – another 255 times. Therefore the maximum number of iterations is in fact 256, with the loop counter initially 0.

So for the longest possible single loop delay, we can write something like:

```

        clrfsz   dc1           ; loop 256 times
dly1    nop
        decfsz  dc1, f
        goto    dly1

```

The two “move” instructions have been replaced with a single 'clrfsz', using 1 cycle less, so the total time taken is $256 \times 4 = 1024$ cycles ≈ 8 ms.

That's still well short of the 0.5 s needed, so we need to wrap (or *nest*) this loop inside another, using separate counters for the inner and outer loops, as shown:

```

        movlw   .61           ; loop (outer) 61 times
        movwf  dc2
        clrfsz  dc1           ; loop (inner) 256 times
dly1    nop                 ; inner loop = 256 x 4 - 1 = 1023 cycles
        decfsz dc1, f
        goto   dly1
        decfsz dc2, f
        goto   dly1

```

The loop counter 'dc2' is being used to control how many times the inner loop is executed.

Note that there is no need to clear the inner loop counter (dc1) on each iteration of the outer loop, because every time the inner loop completes, $dc1 = 0$.

The total time taken for each iteration of the outer loop is 1023 cycles for the inner loop, plus 1 cycle for the 'decfsz dc2, f' and 2 cycles for the 'goto' at the end, except for the final iteration, which, as we've seen, takes 1 cycle less. The three setup instructions at the start add 3 cycles, so if the number of outer loop iterations is N:

Total delay time = $(N \times (1023 + 3) - 1 + 3)$ cycles = $(N \times 1026 + 2) \times 8 \mu\text{s}$.

The maximum delay would be with $N = 256$, giving 2.101 sec, which is more than we need.

With $N = 60$, the delay time = 492.5 ms. Or if $N = 61$, the delay time = 500.7 ms. Either way the delay is around 2% out.

Can we do better if we remove the 'nop'?

The delay code would then be:

```

        movlw   .81           ; loop (outer) 81 times
        movwf  dc2
        clrfsz  dc1           ; loop (inner) 256 times
dly1    decfsz  dc1, f        ; inner loop = 256 x 3 - 1 = 767 cycles
        goto   dly1
        decfsz dc2, f
        goto   dly1

```

The delay time then = $(N \times (767 + 3) - 1 + 3)$ cycles = $(N \times 770 + 2) \times 8 \mu\text{s}$.

With $N = 81$ the delay time = 499.0 ms.

We could try to fine-tune this further, by adding ‘nop’s to the inner loop and adjusting the number of outer loop iterations. For even finer control, you can add ‘nop’s to the outer loop, immediately before the ‘decfsz dc2,f’ instruction. With a bit of fiddling, once you get some nested loops close to the delay you need, adding or removing ‘nop’ or ‘goto \$+1’ instructions can generally get you quite close to the delay you need.

However, it is pointless to aim for high precision (< 1%) when using the internal RC oscillator. When using a crystal, it makes more sense to count every last cycle accurately, as we’ll see in lesson 7.

The code above, providing a delay of 499 ms, is within 0.2% of the desired 500 ms, so there is no point trying for more accuracy than that in this example.

For longer delays, you can select a slower processor clock speed, and/or add more levels of nesting to your delay loops – with enough levels you can count for years!

Of course, before we can use the delay loop counters, dc1 and dc2, we have to allocate data memory for them:

```
;***** VARIABLE DEFINITIONS
          UDATA
dc1      res 1           ; delay loop counters
dc2      res 1
```

As they are 8-bit variables (a single 8-bit register each), we reserve one byte of data memory for each.

Note that, because they will be placed in general purpose RAM, which is banked, it is necessary to use `banksel` before accessing them. But there is no need to use `banksel` every time; it must be used the first time a group of variables is accessed, but not subsequently – unless another bank has been selected.

We know that both variables will be in the same bank, since they are declared as part of the same data section. If you select the bank for one variable in a data section, then it will also be the correct bank for every other variable in that section, so we only need to use `banksel` once. You only need another `banksel` if you access a special function register (potentially changing the bank selection) between variable accesses, or if you access a variable in a different data section.

To avoid the need for banking, you could instead place the variables in common RAM. As explained in [lesson 1](#), not all RAM is banked: “common RAM” is mapped into every bank, meaning that data stored in common RAM can be accessed without having to change the bank selection.

The `UDATA_SHR` directive is used to declare a section of common RAM, or *shared* data memory.

It’s used in the same way as `UDATA`; the only difference is that memory reserved in a `UDATA_SHR` section won’t be banked.

Since only 16 bytes of common RAM is available, it should be used sparingly. However, it can make sense to allocate it for variables that are likely to be used often.

Although we only have a couple of variables in this example, it’s good to get into the habit of using common RAM only when you have a compelling reason to. So we’ll generally continue to use the `UDATA` directive to declare our variables in the more-plentiful banked data memory.

To summarise:

- The first time you access a variable declared in a `UDATA` section, use `banksel`.
- To access subsequent variables in the same `UDATA` section, you don’t need to use `banksel`. (unless you had selected another bank between variable accesses)
- To access variables in a `UDATA_SHR` section, there is never any need to use `banksel`.

Finally, we should select the oscillator frequency, as part of the initialisation routine:

```

; configure oscillator
movlw  b'00111010'      ; configure internal oscillator:
; -0111---             500 kHz (IRCF = 0111)
; -----1-           select internal clock (SCS = 1x)
banksel OSCCON          ; -> 8 us / instruction cycle
movwf  OSCCON

```

Although we are use the default 500 kHz clock, it's good practice to explicitly initialise the oscillator in any program, such as this one, which assumes a specific processor frequency – your code will be more likely to work (or at least you'll see more easily what has to be changed) if you later move it to another processor.

And note the way that this has been commented: the line with '-0111---' makes it clear that only bits 3-6 are relevant to selecting the internal oscillator frequency, and the line with '-----1-' shows that only bit 2 is needed to select the internal oscillator as the clock source.

Complete program

Putting together all these pieces, here's the complete LED flashing program:

```

;*****
;
; Description:      Lesson 2, example 1
;
; Flashes an LED at approx 1 Hz.
; LED continues to flash until power is removed.
;
;*****
;
; Pin assignments:
;   RA1 = flashing LED
;
;*****
#include "p12F1501.inc"

errorlevel  -302          ; no warnings about registers not in bank 0
errorlevel  -303          ; no warnings about program word too large

;***** CONFIGURATION
;   ext reset, internal oscillator (no clock out), no watchdog,
;   brownout resets on, no power-up timer, no code protect
;   no write protection, stack resets on, low brownout voltage,
;   no low-power brownout reset, high-voltage programming
__CONFIG __CONFIG1, _MCLRE_ON & _FOSC_INTOSC & _CLKOUTEN_OFF & _WDTE_OFF &
_BOREN_ON & _PWRTE_OFF & _CP_OFF
__CONFIG __CONFIG2, _WRT_OFF & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF

;***** VARIABLE DEFINITIONS
        UDATA
dc1     res 1              ; delay loop counters
dc2     res 1

;***** RESET VECTOR *****
RES_VECT CODE 0x0000      ; processor reset vector

```

```

;***** MAIN PROGRAM *****
;***** Initialisation
start
    ; configure port
    banksel LATA                ; start with all output pins low
    clrf LATA                   ; (LED off)
    movlw b'111101'            ; configure RA1 (only) as an output
    banksel TRISA
    movwf TRISA

    ; configure oscillator
    movlw b'00111010'          ; configure internal oscillator:
    ; -0111---                500 kHz (IRCF = 0111)
    ; -----1-                select internal clock (SCS = 1x)
    banksel OSCCON              ; -> 8 us / instruction cycle
    movwf OSCCON

;***** Main loop
main_loop
    ; toggle LED
    banksel LATA
    movlw b'000010'            ; toggle LATA bit corresponding to RA1
    xorwf LATA,f

    ; delay 500 ms
    banksel dc1                ; outer loop: 81 x (767 + 3) + 3
    movlw .81                  ; = 62,373 cycles
    movwf dc2                  ; = 499.0 ms @ 8 us/cycle
    clrf dc1                   ; inner loop: 256 x 3 - 1
dly1    decfsz dc1,f           ; = 767 cycles
        goto dly1
    decfsz dc2,f
    goto dly1

    ; repeat forever
    goto main_loop

END

```

If you follow the programming procedure described in [lesson 1](#), you should now see your LED flashing at something very close to 1 Hz.

Conclusion

It's taken two lessons and dozens of pages to get here, but we finally have a flashing LED!

In this lesson, we built on the first, showing how to base a new project on an existing one, modifying it and adding whatever additional features the new project needs.

We saw how to toggle a pin and select the processor clock speed.

We also saw how to define variables and to use decrement instructions with conditional tests to implement loops, and how to use loops to create delays of any length.

In the next lesson we'll see how to make our programs more modular, so that useful pieces of code such as the 500 ms delay developed here can be easily re-used.

Introduction to PIC Programming

Enhanced Mid-Range Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 3: Introducing Modular Code

[Lesson 2](#) introduced delay loops, which we used in flashing an LED.

Delay loops are an example of useful pieces of code that could be re-used in other applications; you don't want to have to re-invent the wheel (or delay loop!) every time. Or, in a larger application, you may need to use delays in several parts of the program. It would be wasteful to have to include multiple copies of the same code in the one program. And if you wanted to make a change to your delay code, it would be not only more convenient, but less likely to introduce errors, if you only have to change it in one place.

Code that is made up of pieces that can be easily re-used, either within the same program, or in other programs, is called *modular*. You'll save yourself a lot of time if you learn to write re-usable, modular code, which is why it's being covered in such an early lesson, even though these techniques are most useful in larger programs.

In this lesson, we will learn about:

- Subroutines
- Paging
- Relocatable code
- External modules

We'll continue to assume that you're using either the [Gooligum Baseline and Mid-Range PIC Training and Development Board](#) or Microchip's Low Pin Count (LPC) Demo Board, with Microchip's MPLAB X integrated development environment – see [lesson 1](#) for details.

Subroutines

The 500 ms delay routine developed in [lesson 2](#) was placed *inline*, within the main loop.

If you wished to re-use it in another part of the program, you would need to repeat the whole routine, wasting program memory and making the source code longer than it needs to be.

You would also have to take care and ensure that you change all of the references to address labels within the routine, when copying and pasting code, to avoid your code inadvertently jumping back from the copy to the original routine. And if you wished to change the way the routine was implemented, you would have to find and update every instance of it in the program.

The usual way to use the same routine in a number of locations in a program is to place it into a *subroutine*.

If we implemented the 500 ms delay as a subroutine, the main loop of the “flash an LED” program would look something like:

```
main_loop
    ; toggle LED
    banksel LATA
    movlw   b'000010'      ; toggle LATA bit corresponding to RA1
    xorwf  LATA,f

    ; delay 500 ms
    call   delay500       ; delay 500ms

    ; repeat forever
    goto  main_loop
```

The ‘call’ instruction – “**call** subroutine” – is similar to ‘goto’, in that it jumps to another program address. But first, it copies (or *pushes*) the address of the next instruction onto the stack.

The *stack* is a set of registers, used to hold the return addresses of subroutines. When a subroutine is finished, the *return address* is copied (*popped*) from the stack to the program counter, and program execution continues with the instruction following the subroutine call.

The enhanced mid-range PICs have 16 stack registers, so a maximum of 16 return addresses can be stored. This means that you can call a subroutine from within another subroutine, which in turn calls yet another subroutine (and so on), but you can’t *nest* the subroutine calls any deeper than 16 levels. However, for the sort of programs you’ll want to write on an 8-bit PIC, you’ll find this isn’t usually a problem. If it is, then it’s probably time to move up to a 16- or 32-bit device...

The instruction to *return* from a subroutine is ‘return’ – “**return** from subroutine”, which, as the name suggests, simply finishes the subroutine by returning to the calling code.

Here then is our 500 ms delay routine, written as a subroutine:

```
delay500                                ; delay 500ms
    banksel dc1                          ; outer loop: 81 x (767 + 3) + 3
    movlw  .81                            ; = 62,373 cycles
    movwf  dc2                            ; = 499.0 ms @ 8 us/cycle
    clrf   dc1                            ; inner loop: 256 x 3 - 1
dly1    decfsz dc1,f                      ; = 767 cycles
    goto  dly1
    decfsz dc2,f
    goto  dly1

    return
```

Parameter Passing with W

A re-usable 500 ms delay routine is all very well, but it’s only useful if you need a delay of 500 ms. What if you want a 200 ms delay – write another routine? Have multiple delay subroutines, one for each delay length? It’s more useful to have a single routine that can provide a range of delays. The requested delay time would be passed as a *parameter* to the delay subroutine.

If you had a number of parameters to pass (for example, a ‘multiply’ subroutine would have to be given the two numbers to multiply), you’d need to place the parameters in general purpose RAM, accessed by both the calling program and the subroutine. But if there is only one parameter to pass, it’s often convenient to simply place it in W.

For example, in the delay routine above, we could simply remove the ‘movlw .81’ line, and instead pass this number (81) as a parameter:

```
movlw    .81
call    delay           ; delay 81 x 6.16ms = 500ms
```

But passing a value of ‘81’ to specify a delay of 500 ms is a little obscure. It would be better if the delay subroutine worked in multiples of an easier-to-use duration than 6.16 ms.

Ideally, we’d pass the number of milliseconds wanted, directly, i.e. pass a parameter of ‘500’ for a 500 ms delay. But that won’t work. The enhanced mid-range PICs are 8-bit devices; the largest value you can pass in any single register, including W, is 255.

If the delay routine produces a delay which is some multiple of 10 ms, it could be used for any delay from 10 ms to 2.55 s, which is quite useful – you’ll find that you commonly want delays in this range.

To implement a $W \times 10$ ms delay, we need an inner loop which creates a 10 ms (or close enough) delay, and an outer loop which counts the specified number of those 10 ms loops.

For example:

```
delay10                                ; delay = 2+Wx(223+1023+4)-1+4
    banksel dc1                          ; = W x 1250 + 5 cycles
    movwf dc2                             ; = W x 10.0 ms @ 8 us/cycle
dly3  movlw .74                           ; inner loop 1: 2 + 74 x 3 - 1
    movwf dc1                             ; = 223 cycles
dly1  decfsz dc1,f
    goto dly1
dly2  nop                                 ; inner loop: 256 x 4 - 1
    decfsz dc1,f                          ; = 1023 cycles
    goto dly2
    nop
    decfsz dc2,f                          ; end outer loop
    goto dly3
return
```

Example 1: Flash an LED (using delay subroutine with parameter passing)

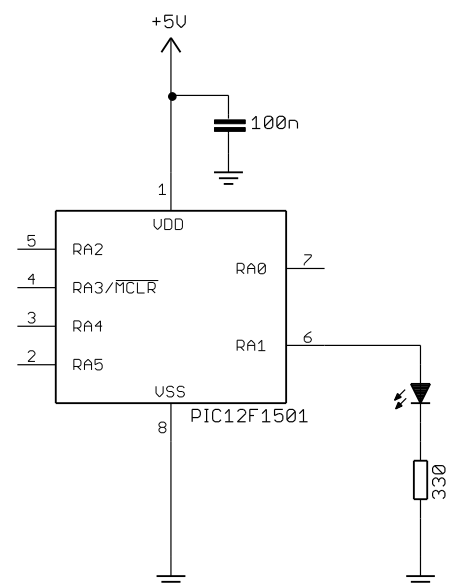
To illustrate where subroutines and parameter passing are useful, suppose that, instead of the LED being on half the time (a 50% *duty cycle*), we want the LED to flash briefly, for say 200 ms, once per second (a 20% *duty cycle*).

That would require a delay of 200 ms while the LED is on, followed by a delay of 800 ms while it is off.

We’ll demonstrate this with the circuit used in the first two lessons, as shown on the right.

If you have the Gooligum training board, simply plug the PIC12F1501 into the top section of the 14-pin IC socket – the section marked ‘12F’. And as before, place a shunt in jumper JP12, to enable the LED on RA1.

If you have the Microchip Low Pin Count Demo board, refer back to [lesson 1](#) to see how to build this circuit, either by soldering a resistor and to the demo board, or by making connections on the demo board’s 14-pin header.



The program will be much the same as that in [lesson 2](#), with the processor configuration and initialisation sections unchanged.

Using the ‘delay10’ subroutine presented above, the main loop becomes:

```
main_loop
    ; turn on LED
    banksel LATA
    bsf     LATA,RA1          ; RA1 -> high

    ; delay 0.2 s
    movlw  .20                ; delay 20 x 10 ms = 200 ms
    call   delay10

    ; turn off LED
    banksel LATA
    bcf     LATA,RA1          ; RA1 -> low

    ; delay 0.8 s
    movlw  .80                ; delay 80 x 10 ms = 800 ms
    call   delay10

    ; repeat forever
    goto   main_loop
```

Complete program

Here is the complete program to flash an LED at 1 Hz with a 20% duty cycle, illustrating how the above pieces fit together.

You’ll see that the subroutine has been placed into a “SUBROUTINES” section toward the end, and clearly documented – if you’re using subroutines in your code, it’s good to be able to easily find them and see what they do, in case you’ve forgotten, or if you want to re-use a subroutine in another program:

```
*****
;
; Description:      Lesson 3, example 1
;
; Flashes an LED at approx 1 Hz, with 20% duty cycle
; LED continues to flash until power is removed.
;
; Uses W x 10 ms delay subroutine
;
*****
;
; Pin assignments:
; RA1 = indicator LED
;
*****

#include "p12F1501.inc"

errorlevel -302          ; no warnings about registers not in bank 0
errorlevel -303          ; no warnings about program word too large

;***** CONFIGURATION
; ext reset, internal oscillator (no clock out), no watchdog,
; brownout resets on, no power-up timer, no code protect
; no write protection, stack resets on, low brownout voltage,
```

```

        ; no low-power brownout reset, high-voltage programming
__CONFIG __CONFIG1, _MCLRE_ON & _FOSC_INTOSC & _CLKOUTEN_OFF & _WDTE_OFF &
_BOREN_ON & _PWRTE_OFF & _CP_OFF
__CONFIG __CONFIG2, _WRT_OFF & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF

```

```

;***** VARIABLE DEFINITIONS

```

```

        UDATA
dc1      res 1          ; delay loop counters
dc2      res 1

```

```

;***** RESET VECTOR *****
RES_VECT CODE    0x0000          ; processor reset vector

```

```

;***** MAIN PROGRAM *****

```

```

;***** Initialisation
start

```

```

        ; configure port
        movlw    ~(1<<RA1)      ; configure RA1 (only) as an output
        banksel TRISA
        movwf   TRISA

        ; configure oscillator
        movlw    b'00111010'    ; configure internal oscillator:
        ; -0111---              500 kHz (IRCF = 0111)
        ; -----1-              select internal clock (SCS = 1x)
        banksel OSCCON          ; -> 8 us / instruction cycle
        movwf   OSCCON

```

```

;***** Main loop

```

```

main_loop
        ; turn on LED
        banksel LATA
        bsf     LATA,RA1        ; RA1 -> high

        ; delay 0.2 s
        movlw   .20             ; delay 20 x 10 ms = 200 ms
        call   delay10

        ; turn off LED
        banksel LATA
        bcf     LATA,RA1        ; RA1 -> low

        ; delay 0.8 s
        movlw   .80             ; delay 80 x 10 ms = 800 ms
        call   delay10

        ; repeat forever
        goto   main_loop

```

```

;***** SUBROUTINES *****

```

```

;***** Variable delay: 10 ms to 2.55 s

```

```

;
; Delay = W x 10 ms
;

```

```

delay10                ; delay = 2+Wx(223+1023+4)-1+4
    banksel dc1         ;   = W x 1250 + 5 cycles
    movwf dc2          ;   = W x 10.0 ms @ 8 us/cycle
dly3    movlw .74      ; inner loop 1: 2 + 74 x 3 - 1
    movwf dc1         ;   = 223 cycles
dly1    decfsz dc1,f
    goto dly1
dly2    nop           ; inner loop: 256 x 4 - 1
    decfsz dc1,f     ;   = 1023 cycles
    goto dly2
    nop
    decfsz dc2,f     ; end outer loop
    goto dly3

    return

    END

```

Relocatable Modules

If you wanted to take a subroutine you had written as part of one program, and re-use it in another, you could simply copy and paste the source code into the new program.

There are a few potential problems with this approach:

- Address labels, such as 'dly1', may already be in use in the new program or in other pieces of code that you're copying.
- You need to know which variables are needed by the subroutine, and remember to copy their definitions to the new program.
- Variable names have the same problem as address labels – they may already be used in new program, in which case you'd need to identify and rename all references to them.
- The subroutine may need a particular include file; this will need to be identified and included in the new program.

These *namespace* clashes and other problems can be avoided by keeping the subroutine code in a separate source file, where it is assembled into an object file, called an *object module*.

The main code is assembled into a separate object file. These object files – one for the main code, plus one for each module, are then linked together to create the final executable code, which is output as a .hex file to be programmed into the PIC. This assembly/link (or *build*) process sounds complicated, but MPLAB takes care of the details, as we'll see later.

To be relocatable, a module must have its own data sections, to keep its variables separate from the rest of the program's variables. The linker can place these data sections anywhere in data memory – perhaps in a different bank from your other variables.

When you are using more than one data section, which will usually be the case if you are using relocatable modules, you must ensure that you set the bank selection bits correctly when accessing variables.

A relocatable module must also have its own code sections, which the linker can place anywhere in memory (hence the term 'relocatable'). This presents a potential problem, as we'll see in the next section on paging.

Paging

As we saw in [lesson 1](#), enhanced mid-range PIC instructions are 14 bits wide and consist of an opcode, designating the instruction, with the remaining bits specifying the a value, such as a register address.

The opcode for `goto` is 101, while that for `call` is 100. Each is three bits, leaving eleven bits to specify the address to jump to.

Eleven bits are enough to specify any value from 0 to 2047. That's 2048, or 2k, addresses in all.

This is called a *page* of program memory.

The program memory on the 12F1501 is 1024 words, which is half a page. Since the `goto` instruction can directly specify any of these 1024 addresses, it can be used to jump anywhere in the 12F1501's memory.

That's fine for the 12F1501, but it's a problem for a larger device such as the 16F1824 with 4096 (4k) words or 16F1518 with 16k words. In fact, the enhanced mid-range PIC architecture provides for up to 32k words of program memory. How can a `goto` or `call` instruction, which can only directly specify an address within a single 2k page, access this larger program memory space?

The solution is to use the `PCLATH` register to select which page is to be accessed.

The *program counter* (`PC`) holds the full 15-bit address of the next instruction to be executed.

Whenever a `goto` or `call` instruction is executed, the lower 11 bits of the program counter (`PC<10:0>`) are taken from the instruction word, but the upper bits (`PC<14:11>`) are copied from bits 3 to 6 of the `PCLATH` register¹.

If you don't update `PCLATH` before you try to `goto` or `call` an address in a different page, you will instead jump to the corresponding address in the current page – not the location you were trying to access, and your program will almost certainly fail.

To make this easy to do, the assembler provides a `'pagesel'` directive, which instructs the assembler and linker to generate code to select the correct page for the given program address².

Page selection is relevant to a discussion of modular code, because the linker may load an object module anywhere in memory; that is why these modules, and this programming style, are described as being *relocatable*. This means that, when calling a subroutine in another module, you will not know if the subroutine is in the current page.

This is also true if you use multiple `CODE` sections within a single source file; unless you place the code sections at a specific address (which is not recommended, since it makes it more difficult for the linker to fit the sections into memory pages), you cannot know where each section will be placed in memory.

Therefore, you should use `pagesel` whenever jumping to or calling a routine in a different code section or module. And note that, after returning from a call to a module, the page selection bits will still be set for whatever page that module is in, not necessarily the current page. So it is a good idea to place a `'pagesel $'` directive (“select page for current address”) after each `call` to a subroutine in another module, to ensure that the current page is selected after returning from the subroutine.

¹ The lower bits of `PCLATH` are used when a “computed goto” or “computed function call” operation is performed, as we will see in future lessons.

² It is somewhat similar to the `banksel` directive used to select the appropriate bank before accessing data memory.

You do not, however, need to use `pagesel` before every `goto` or `call`, or after every `call`. Each `CODE` section is guaranteed to be wholly contained within a single page³. Once you know that you've selected the correct page, subsequent `gotos` or `calls` to the same section will work correctly. But be careful!

If in doubt, using `pagesel` before every `goto` and `call` is a safe approach that will always work.

When assembling code for a device, such as the PIC12F1501, which has only a single page of program memory, the `pagesel` directive will not generate any object code, so there is no penalty for using it on PICs where page selection is not an issue. The assembler will, however, warn you that `pagesel` isn't needed on these devices. If you find these messages annoying, you can turn them off with:

```
errorlevel -312 ; no "page or bank selection not needed" messages
```

If you use `pagesel`, even on devices with only a single page of program memory, your code will be more portable, so it is best to always use it, regardless of which enhanced mid-range PIC you are using.

Creating a Relocatable Module

Converting an existing subroutine, such as the 'delay10' routine, into a standalone, relocatable module is easy. All you need to do is to declare any symbols (address labels or variables) that need to be accessible from other modules, using the `GLOBAL` directive. For example:

```
#include "p12F1501.inc" ; any enhanced mid-range device will do

GLOBAL delay10

;***** VARIABLE DEFINITIONS
        UDATA
dc1     res 1 ; delay loop counters
dc2     res 1

;***** SUBROUTINES *****
        CODE

;***** Variable delay: 10 ms to 2.55 s
;
; Delay = W x 10 ms
;
delay10 ; delay = 2+Wx(223+1023+4)-1+4
        banksel dc1 ; = W x 1250 + 5 cycles
        movwf dc2 ; = W x 10.0 ms @ 8 us/cycle
dly3    movlw .74 ; inner loop 1: 2 + 74 x 3 - 1
        movwf dc1 ; = 223 cycles
dly1    decfsz dc1,f
        goto dly1
dly2    nop ; inner loop: 256 x 4 - 1
        decfsz dc1,f ; = 1023 cycles
        goto dly2
        nop
        decfsz dc2,f ; end outer loop
        goto dly3

        return

        END
```

³ unless you are an advanced PIC developer and create your own linker scripts...

This is the subroutine from example 1, with a `CODE` directive at the beginning of it, and a `UDATA` directive to reserve data memory for the subroutine's variables.

Toward the start, a `GLOBAL` directive has been added, declaring that the `'delay10'` label is to be made available (*exported*) to other modules, so that they can call this subroutine.

You should also add a `'#include'` directive, to define any “standard” symbols used in the code, such as the instruction destinations `'w'` and `'f'`. This delay routine will work on any enhanced mid-range PIC; it's not specific to any, so you can use the include file for any of the mid-range PICs, such as the 12F1501. You will, however, see warnings about “Processor-header file mismatch” if your device is different to the processor that the include file is intended for; you can generally ignore these warnings, but, if in doubt, change the `'#include'` directive in the module to match the processor you are building the code for.

Of course it's also important to add a block of comments at the start; they should describe what this module is for, how it is used, any effects (including side effects) it has, and any assumptions that have been made. In this case, it is assumed that the processor is clocked at the default 500 kHz, so that assumption should be documented in the comment block.

Calling Relocatable Modules

Having created an *external* relocatable module (i.e. one in a separate file), we need to declare, in the main (or *calling*) file any labels we want to use from the external module, so that the linker knows that these labels are defined in another module. That's done with the `EXTERN` directive.

For example:

```
EXTERN      delay10          ; W x 10ms delay
```

After having been declared as external, it is then possible to call a subroutine or access a variable in an external module (using `pagesel` or `banksel` first!) in the usual way.

To summarise:

- The `GLOBAL` and `EXTERN` directives work as a pair.
- `GLOBAL` is used in the file that defines a module, to export a symbol for use by other modules.
- `EXTERN` is used when calling external modules. It declares that a symbol has been defined elsewhere.

Example 2: Flashing an LED (using an external module)

As we did in [lesson 2](#), we can use the circuit from example 1 above to flash an LED at 1 Hz, with a 50% duty cycle – but this time using an external delay module. We'll then call this external module from the main program.

We'll setup a project with the following files:

- `delay10-enh.asm` - containing the $W \times 10$ ms delay module, as above
- `EA_L3_2-Flash_LED-50p-mod.asm` - the main code (calling the delay routine)

(or whatever names you choose)

Creating a multiple-file project, using MPLAB X

To create the multiple-file project, open an existing project and then copy it with a new name, such as “EA_L3_2-Flash_LED-50p-mod”, in the same way as you did when creating new project in [lesson 2](#).

Rename the source file to “EA_L3_2-Flash_LED-50p-mod.asm” (for example), in the same way as was done in [lesson 2](#) – right-click it in the Projects window and select “Rename...”.

Next we need to copy this file, creating a new file which will contain our delay module.

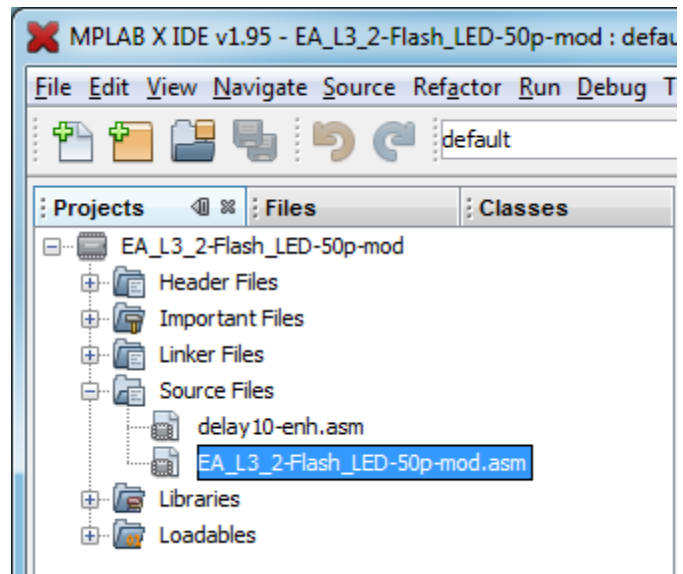
There are a few ways to do this, but the easiest is probably to right-click the source file in the Projects window and select “Copy”.

Right-click “Source Files” in the project tree, and select “Paste”.

A new .asm file (a copy of the original) should appear in the project tree.

You can now right-click this new file, and rename it to “delay10-enh.asm”.

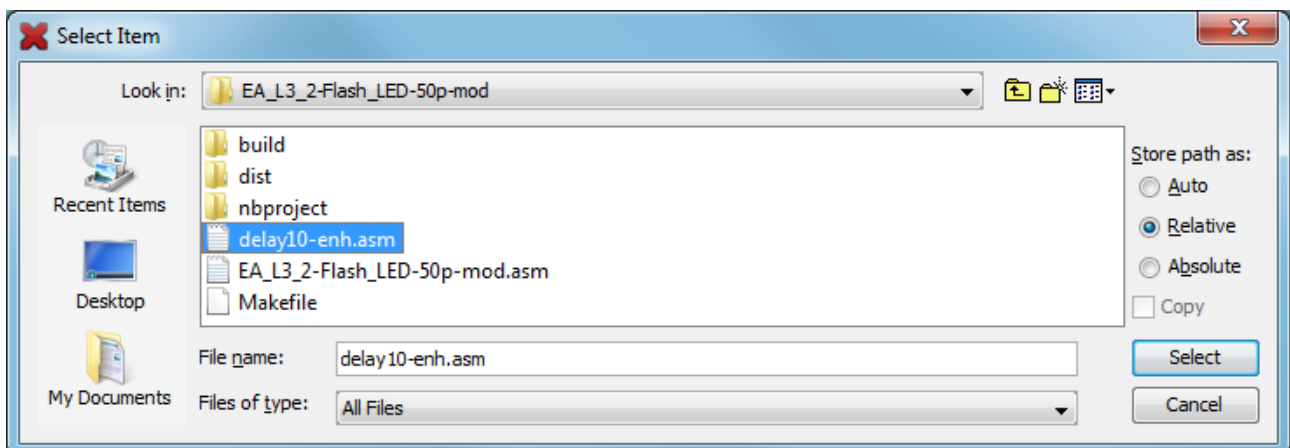
Your project should look like the one shown on the right.



Another way to do this is to double-click the original source file (the one you want to copy), opening an editor window. If you now activate the editor window, by clicking anywhere in it, you can use the “File → Save As...” menu item to save the file as “delay10.asm”.

The only problem is that this new source file hasn’t appeared in the Projects window; MPLAB X doesn’t yet know that the new file is part of the project. So, we need to add it.

To add an existing file (or files) such as one that you have copied, or the example source code files provided with these tutorials, to a project, you can right-click on “Source Files” in the Projects window, and then select “Add Existing Item...”. You will be presented with the window shown below:



As you can see, it gives you the option to specify whether the file has a relative path (appropriate for most “user” files) or absolute path (for most “system” files). If you’re unsure, just select “Auto” and let MPLAB decide.

If you want to create a new file from scratch, instead of using an existing one, you can use the “File → New File...” menu item, in the same way as we did in [lesson 1](#).

However you created them, now that you have a project which includes the two source files, we can consider their content...

We've already seen the code for the "delay10" module, presented above, but here it is again with comments, as the complete "delay10-enh.asm" file:

```

;*****
; Architecture:   Enhanced Mid-range PIC          *
; Processor:     any                             *
;                                                         *
;*****
; Files required: none                            *
;                                                         *
;*****
; Description:   Variable Delay : N x 10 ms (10 ms - 2.55 s) *
;                                                         *
;               N passed as parameter in W reg      *
;               exact delay = W x 10.0 ms + 40 us   *
;                                                         *
; Affects: W, STATUS, BSR                          *
; Assumes: 500 kHz clock                            *
;*****

#include "p12F1501.inc"      ; any enhanced mid-range device will do

GLOBAL      delay10

;***** VARIABLE DEFINITIONS
          UDATA
dc1       res 1              ; delay loop counters
dc2       res 1

;***** SUBROUTINES *****
          CODE

;***** Variable delay: 10 ms to 2.55 s
;
; Delay = W x 10 ms
;
delay10          ; delay = 2+Wx(223+1023+4)-1+4
                ;   = W x 1250 + 5 cycles
                ;   = W x 10.0 ms @ 8 us/cycle
dly3      movlw   .74        ; inner loop 1: 2 + 74 x 3 - 1
                ;   = 223 cycles
dly1      decfsz  dc1,f
                goto   dly1
dly2      nop              ; inner loop: 256 x 4 - 1
                ;   = 1023 cycles
                decfsz  dc1,f
                goto   dly2
                nop
                decfsz  dc2,f      ; end outer loop
                goto   dly3

          return

          END

```

We now need to modify the main (or *calling*) file by changing the code to call the external 'delay10' module, as shown:

```

;*****
;
; Architecture: Enhanced Mid-range PIC
; Processor: 12F1501
;
;*****
;
; Files required: delay10-enh.asm (provides W x 10 ms delay)
;
;*****
;
; Description: Lesson 3, example 2
;
; Demonstrates how to call external modules
;
; Flashes an LED at approx 1 Hz.
; LED continues to flash until power is removed.
;
; Uses W x 10 ms delay module
;
;*****
;
; Pin assignments:
; RA1 = indicator LED
;
;*****

#include "p12F1501.inc"

errorlevel -302 ; no warnings about registers not in bank 0
errorlevel -303 ; no warnings about program word too large
errorlevel -312 ; no "page or bank selection not needed" messages

EXTERN delay10 ; W x 10ms delay

;***** CONFIGURATION
; ext reset, internal oscillator (no clock out), no watchdog,
; brownout resets on, no power-up timer, no code protect
; no write protection, stack resets on, low brownout voltage,
; no low-power brownout reset, high-voltage programming
__CONFIG __CONFIG1, _MCLRE_ON & _FOSC_INTOSC & _CLKOUTEN_OFF & _WDTE_OFF &
_BOREN_ON & _PWRTE_OFF & _CP_OFF
__CONFIG __CONFIG2, _WRT_OFF & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF

;***** RESET VECTOR *****
RES_VECT CODE 0x0000 ; processor reset vector

;***** MAIN PROGRAM *****

;***** Initialisation
start
; configure port
movlw ~(1<<RA1) ; configure RA1 (only) as an output
banksel TRISA
movwf TRISA

```

```

; configure oscillator
movlw    b'00111010'    ; configure internal oscillator:
; -0111---            500 kHz (IRCF = 0111)
; -----1-          select internal clock (SCS = 1x)
banksel  OSCCON        ; -> 8 us / instruction cycle
movwf   OSCCON

;***** Main loop
main_loop
; toggle LED
banksel  LATA
movlw   1<<RA1        ; toggle LATA bit corresponding to RA1
xorwf   LATA,f

; delay 500 ms -> 1 Hz flashing at 50% duty cycle
movlw   .50
pagesel delay10      ; delay 50 x 10 ms = 500 ms
call    delay10

; repeat forever
pagesel main_loop
goto   main_loop

END

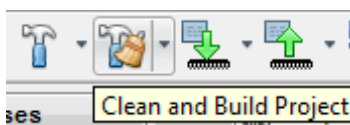
```

The inline delay routine has been replaced with a call our external delay module, and the variables used by the delay routine removed. And toward the start of the program, an `EXTERN` directive has been added, to declare that the ‘`delay10`’ label is a reference to another module.

We’ve also added `pagesel` directives to ensure that the code would work correctly if moved to a device with more than one page of program memory. And, because the `pagesel` directives aren’t actually required on a PIC12F1501 (which only has one page of program memory), we’ve added the “`errorlevel -312`” directive to avoid generating warnings about page selection not being needed.

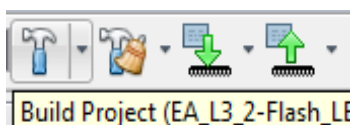
Also note that we’ve documented, in the comments block at the start of the source code, the fact that this program relies on an external module, what that module does, and what file it is defined in.

The Build Process (Revisited)



In a multiple-file project, when you select “Run → Clean and Build” or click on the “Clean and Build” toolbar button, the assembler will assemble all the source files, producing a new ‘.o’ *object file* for each. The linker then combines these ‘.o’ files to build a single ‘.hex’ file, containing an image of the executable code to be programmed into the PIC.

If, however, you’ve been developing a multi-file project, and you’ve already built it, and then go back and alter just one of the source files, there’s no need to re-assemble all the other source files, if they haven’t changed.



That’s what the “Run → Build” menu item or the “Build” toolbar button does, as was discussed briefly in [lesson 1](#). Like “Clean and Build”, it builds your project, but only assembles source files which have a newer date stamp than the corresponding object file. This is what you normally want, to save unnecessary assembly time (not that it makes much difference with such a small project!).

After you build (or make) the project, you'll see a number of new files in various folders within the project directory. In addition to your '.asm' source files and the '.o' object files and the '.hex' output file we've already discussed, you'll find '.lst' files corresponding to each of the source files, and a '.map' file corresponding to the project name.

There is no need to describe these in detail, but they are worth looking at if you are curious about the build process. And they can be valuable to refer to if you when debugging, as they show exactly what the assembler and linker are doing.

The '.lst' *list files* show the output of the assembler; you can see the opcodes corresponding to each instruction. They also show the value of every label. But you'll see that, for the list files belonging to the source files (e.g. "delay10-enh.lst"), they contain a large number of question marks. For example:

```
0000          00049 delay10          ; delay = 2+Wx(223+1023+4)-1+4
0000 002?          00050      banksel dc1          ; = W x 1250 + 5 cycles
0001 00??          00051      movwf dc2          ; = W x 10.0 ms @ 8 us/cycle
0002 304A          00052 dly3      movlw .74          ; inner loop 1: 2 + 74 x 3 - 1
0003 00??          00053      movwf dc1          ; = 223 cycles
0004 0B??          00054 dly1      decfsz dc1,f
0005 2???          00055      goto dly1
0006 0000          00056 dly2      nop          ; inner loop: 256 x 4 - 1
0007 0B??          00057      decfsz dc1,f      ; = 1023 cycles
0008 2???          00058      goto dly2
0009 0000          00059      nop
000A 0B??          00060      decfsz dc2,f      ; end outer loop
000B 2???          00061      goto dly3
          00062
000C 0008          00063      return
```

Many of the instruction opcodes are only partially complete. The question marks can't be filled in, until the locations of all the data and program labels, such as 'dc1' and 'dly1', are known.

Assigning locations to the various objects is the linker's job, and you can see the choices it has made by looking at the project's '.map' *map file*. It shows where each section will be placed, and what the final data and program addresses are. For example (reformatted a little here):

		Section Info			
	Section	Type	Address	Location	Size (Bytes)
	RES_VECT	code	0x000000	program	0x000018
	.cinit	romdata	0x00000c	program	0x000004
	.code	code	0x00000e	program	0x00001a
	.config_8007_BUILD/DEFAULT/PRODUCTION/EA_L3_2	code	0x008007	program	0x000002
	.config_8008_BUILD/DEFAULT/PRODUCTION/EA_L3_2	code	0x008008	program	0x000002
	.udata	udata	0x000020	data	0x000002

Program Memory Usage
Start End

0x000000 0x00001a
0x008007 0x008008

29 out of 1031 program addresses used, program memory utilization is 2%

Symbols - Sorted by Name				
Name	Address	Location	Storage	File
delay10	0x00000e	program	extern	C:\...\delay10-enh.asm
dly1	0x000012	program	static	C:\...\delay10-enh.asm
dly2	0x000014	program	static	C:\...\delay10-enh.asm
dly3	0x000010	program	static	C:\...\delay10-enh.asm
main_loop	0x000006	program	static	C:\...\EA_L3_2-Flash_LED-50p-mod.asm
start	0x000000	program	static	C:\...\EA_L3_2-Flash_LED-50p-mod.asm
dc1	0x000020	data	static	C:\...\delay10-enh.asm
dc2	0x000021	data	static	C:\...\delay10-enh.asm

These addresses are used when the linker creates the '.hex' file, containing the final assembled code, with fully resolved addresses, that will be loaded into the PIC.

Conclusion

Again, this has been a lot theory – and we’re still only flashing an LED!

The intent of this lesson was to give you an understanding of enhanced mid-range PIC program memory, including its limitations (paging) and how to work around them, to avoid potential problems as your programs grow.

We’ve also seen how to use subroutines and create and call re-usable code modules, which should help you to avoid wasting time “reinventing the wheel” for each new project in future. In fact, we’ll continue to use the delay module in later lessons.

In addition to providing an output (such as a blinking LED), real PIC applications usually involve responding to the environment, or at least to user input.

So, in the next lesson we’ll look at reading and responding to switches, such as pushbuttons.

And since real switches “bounce”, and that can be a problem for microcontroller applications, we’ll look at ways to “debounce” them.

Introduction to PIC Programming

Programming Enhanced Mid-Range PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 1: Light an LED

This series of tutorial lessons introduces the features of enhanced mid-range PICs, using C.

Although assembly language is commonly used when programming small microcontrollers, it is less appropriate for complex applications on larger MCUs; it can become unwieldy and difficult to maintain as programs grow longer. A number of higher-level languages are used in embedded systems development, including BASIC, Forth and even Pascal. But the most commonly used “high level” language is C.

C is often considered to be inappropriate for very small MCUs, such as baseline PICs (see the [baseline assembler](#) and [C](#) tutorial series). However, the enhanced mid-range PIC architecture is much less restrictive and is well suited to C programming, as this tutorial series will demonstrate.

Microchip’s XC8 compiler supports all enhanced mid-range PICs, and can be operated in “Free mode” (with most optimisations disabled) for free – making it a good choice for use in these lessons.

This lesson starts by simply making an LED, connected to one of the output pins of a PIC, light up. This apparently straightforward task – not even flashing the LED at first¹ – relies on:

- Having a functioning circuit in a workable prototyping environment
- Being able to use a development environment; to go from text to compiled C code
- Being able to correctly use a PIC programmer to load the code into the PIC chip
- Correctly configuring the PIC
- Writing code that will make the correct pin output a high or low

If you can get an LED to light up, then you know that you have a development environment that works, and enough understanding of your PIC device to get started. It’s a firm base to build on.

In summary, this lesson covers:

- Introduction to the Microchip XC8 compiler
- Introduction to the enhanced mid-range PIC architecture, using the PIC12F1501
- Simple control of digital output pins
- Using MPLAB X and XC8 to create C projects
- Using a PICkit 3 programmer with MPLAB X

These tutorials assume a working knowledge of the C language; they **do not** attempt to teach C.

¹ We’ll get to the traditional first exercise in microcontroller programming of flashing an LED in [lesson 2](#)...

Getting Started

For some background on PICs in general and details of the recommended development environment, see [lesson 0](#). Briefly, these tutorials assume that you are using a Microchip PICkit 3 programmer and either the [Gooligum Baseline and Mid-range PIC Training and Development Board](#) or Microchip's Low Pin Count (LPC) Demo Board, with Microchip's MPLAB X integrated development environment. But it is of course possible to adapt these instructions to different programmers and/or development boards.

The four LEDs on the LPC demo board don't work (directly) with 8-pin PICs, such as the 12F1501. So to complete this lesson, using an LPC demo board, you need to either add an additional LED and resistor to the prototyping area on your board, or use some solid core hook-up wire to patch one of the LEDs to the appropriate PIC pin, as described later.

This is one reason the Gooligum training board was developed to accompany these tutorials – if you have the Gooligum board, you can simply plug in your 8-pin 12F PIC, and go.

We're going to start with the simplest enhanced mid-range PIC – the PIC12F1501.

Of course, "simplest" is a relative term. The enhanced mid-range architecture is certainly more complex than the earlier baseline PIC architecture, introduced in the [baseline PIC assembler](#) and [C](#) tutorial series. Those lessons were able to start with a very simple PIC indeed (the 10F200), which made it possible to introduce only a few basic topics at first, without needing to say "ignore this for now; we'll explain later". More advanced topics were introduced by moving up to more advanced baseline and then eventually mid-range PICs through the [mid-range PIC assembler](#) and [C](#) tutorial series – building on what came before.

This tutorial series doesn't refer back to those earlier lessons – it's a fresh start. Unfortunately that does make it harder to ignore some of the complexities of the enhanced mid-range architecture, although we'll keep it as simple as possible to begin with. If you do want to start learning with simpler PICs, you should consider working through the [baseline](#) and [mid-range](#) tutorial series.

Luckily for C programmers, much of the complexity of the enhanced mid-range PIC architecture is hidden by the C compiler. When programming in C, we don't need to be aware of how the PIC's memory is arranged, or what many of the "core registers", which affect the device's basic operation, do. Of course, it can be very useful to understand these lower-level details of the PIC's operation, and if you want to gain that more detailed understanding you should refer to the [enhanced mid-range assembler tutorial series](#). But these lessons, using C, do not assume that you have completed the corresponding assembler lessons.

To repeat – the earlier lessons are not a prerequisite for these enhanced mid-range C lessons.

In summary, for this lesson you should ideally have:

- A PC running Windows 7 or 8, with a spare USB port
- Microchip's MPLAB X IDE software and XC8 C compiler
- A Microchip PICkit 3 PIC programmer
- The Gooligum mid-range training board
- A PIC12F1501-I/P microcontroller (supplied with the Gooligum training board)

Introducing the PIC12F1501

When working with any microcontroller, you should always have on hand the latest version of the manufacturer's data sheet. You should download the download the current data sheet for the 12F1501 from www.microchip.com.

The features of various 8-pin PICs are summarised in the following table:

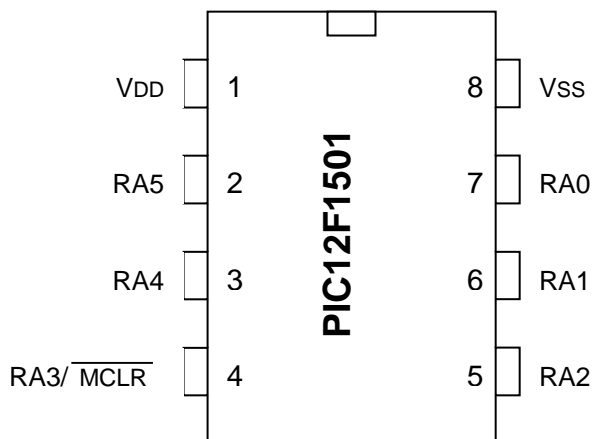
Device	Memory (words or bytes)			Timers		Analog		Clock rate (max MHz)
	Program	Data	EEPROM	8-bit	16-bit	Comp- arators	ADC inputs	
12F508	512	25	0	1	0	0	0	4
12F629	1024	64	128	1	1	1	0	20
12F675	1024	64	128	1	1	1	4	20
12F683	2048	128	256	2	1	1	4	20
12F1501	1024	64	0	2	1	1	4	20
12F1822	2048	128	256	2	1	1	4	32
12F1840	4096	256	256	2	1	1	4	32

Program memory is, as you might expect, where your program is stored. It's non-volatile – it keeps its contents, even when power is removed. Data memory consists of 8-bit bytes where your program stores its data – but it's volatile; the contents of data memory are lost when the device loses power.

We'll look at the various other features mentioned in the table, such as timers, analog inputs, and EEPROM memory, in later lessons. But even without knowing what these things are, you can see that the 12F1501 has fewer features than other enhanced mid-range PICs, such as the 12F1822 or 12F1840, while being roughly comparable to the 12F675 and 12F683 mid-range devices, and significantly more capable than the baseline 12F508.

The 12F family are all 8-pin devices, with six pins available for I/O (input and output).

They share a common pin-out, as shown below.



VDD is the positive power supply.

VSS is the “negative” supply, or ground. All of the input and output levels are measured relative to VSS. In most circuits, there is only a single ground reference, considered to be at 0 V (zero volts), and VSS will be connected to ground.

The power supply voltage on the PIC12F1501 can range from 2.3 V to 5.5 V².

This wide range means that the PIC's power supply can be very simple. Depending on the circuit, you may need no more than a pair of 1.5 V batteries.

Normally you'd place a capacitor, typically 100 nF and ceramic, between VDD and VSS, close to the chip, to smooth transient changes to the power supply voltage caused by changing loads (e.g. motors, or something as simple as an LED turning on) or noise in the circuit.

² A low-power variant, the PIC12LF1501, is also available, where VDD can range from 1.8 V to 3.6 V, with at least 2.5 V needed for clock rates above 16 MHz in both variants.

The remaining pins, RA0 to RA5, are the I/O pins. They are used for digital input and output, except for RA3, which can only be an input. The other pins – RA0, RA1, RA2, RA4 and RA5 – can be individually set to be inputs or outputs.

PIC12F1501 Input and Output

As mentioned above, the 12F1501 has six I/O pins: RA0, RA1, RA2, RA4 and RA5, which can be used for digital input and output, plus RA3, which is input-only.

Taken together, the six I/O pins comprise the general-purpose I/O *port*, or GPIO port, which is referred to as PORTA on the 12F1501.

The PIC's features, such as its I/O ports, are controlled and accessed via 8-bit *registers*. For details of how they are arranged, see the [enhanced mid-range assembler tutorial series](#). But to program PICs in C, we only need to know what the registers are named – and of course how to use them, which often means knowing what the various bits comprising a register are used for.

The PORTA register provides access to the port pins:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTA			RA5	RA4	RA3	RA2	RA1	RA0

If a pin is configured as an output, setting the corresponding PORTA bit to '1' outputs a high voltage³ on the corresponding pin; clearing it to '0' outputs a low voltage⁴.

Reading the PORTA register reads the voltage present on each pin. If the voltage on a pin is high⁵, the corresponding bit reads as '1'; if the input voltage is low⁶, the corresponding bit reads as '0'.

The TRISA register controls whether a pin is configured as an input or output:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TRISA			TRISA5	TRISA4		TRISA2	TRISA1	TRISA0

To configure a pin as an input, set the corresponding bit in the TRISA register to '1'. To make it an output, clear the corresponding TRISA bit to '0'.

Why is it called 'TRIS'? Each pin (except RA3) can be configured as one of three states: high-impedance input, output high, or output low. In the input state, the PIC's output drivers are effectively disconnected from the pin. Another name for an output that can be disconnected is '*tri-state*' – hence, TRIS.

Note that bit 3 of TRISA is greyed-out. Clearing this bit will have no effect, as RA3 is always an input.

The default state for each pin is 'input'; TRIS is set to all '1's when the PIC is powered on or reset.

³ a 'high' output will be within 0.7 V of the supply voltage (VDD), for small pin currents (< 3.5 mA with VDD = 5 V)

⁴ a 'low' output is less than 0.6 V, for small pin currents (< 8 mA with VDD = 5 V)

⁵ the threshold level depends on the power supply, but a 'high' input is any voltage above 2.0 V, given a 5 V supply

⁶ a 'low' input is anything below 0.8 V, given a 5 V supply – see the data sheet for details of each of these levels

It's important to understand that, regardless of whether a pin is configured as an input or an output, PORTA reflects the actual voltage present on that pin.

Note: the port registers represent the actual voltages present on each digital I/O pin, including pins configured as digital outputs

If you attempt to output a 'high' on an output pin by writing a '1' to the corresponding port bit, but the external circuit holds that pin low, that pin will read as '0' – not what you might have expected.

This behaviour can lead to what are known as *read-modify-write* problems, where instructions which are intended to modify only specific pins actually read the entire port, including pins which may not reflect the value that had been output to them, and then write the new value (with some bits possibly incorrect) back to the port.

To avoid the potential for read-modify-write problems, the enhanced mid-range architecture makes available an "output data latch" register, associated with each port:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
LATA			LATA5	LATA4		LATA2	LATA1	LATA0

Writing to LATA has the same effect as writing to PORTA: if a pin is configured as an output, setting the corresponding LATA bit to '1' outputs a 'high' on that pin; clearing it to '0' outputs a 'low'.

However – reading LATA returns the value that was last written to LATA. It does not read the voltages on the pins (whether input or output) themselves.

This means that you can avoid read-modify-write problems by following these rules:

- if you are writing an entire byte to a port, you can write to either PORTA or LATA
- if you are modifying individual port pins, you should operate on LATA
- if you are reading digital input pins, you must read PORTA

To keep it simpler, you won't run into any problems if you always access LATA to write to or modify output pins, and PORTA to read digital input pins.

This should become clearer later as we work through the examples.

When configured as an output, each I/O pin on the 12F1501 can source or sink (i.e. current into or out of the pin) up to 25 mA – enough to directly drive an LED.

PICs are tough devices, and you may get away with exceeding these limits – but if you ignore the absolute maximum ratings specified in the data sheet, you're on your own. Maybe your circuit will work, maybe not. Or maybe it will work for a short time, before failing. It's better to follow the data sheet...

Introducing the XC8 Compiler

XC8 is a descendant of a compiler originally created by a company called HI-TECH Software, which has since been acquired by Microchip.

XC8 supports the whole 8-bit PIC10/12/16/18 series in a single edition, with different licence keys unlocking different levels of code optimisation – "Free" (free, but very little optimisation), "Standard" and "PRO" (most expensive and highest optimisation).

Microchip XC compilers are also available for the PIC24, dsPIC and PIC32 families.

XC8's "Free mode" supports all 8-bit (including baseline, mid-range and enhanced mid-range) PICs, with no memory restrictions. However, in this mode, most compiler optimisation is turned off, making the generated code around twice the size of that generated by the "PRO" version.

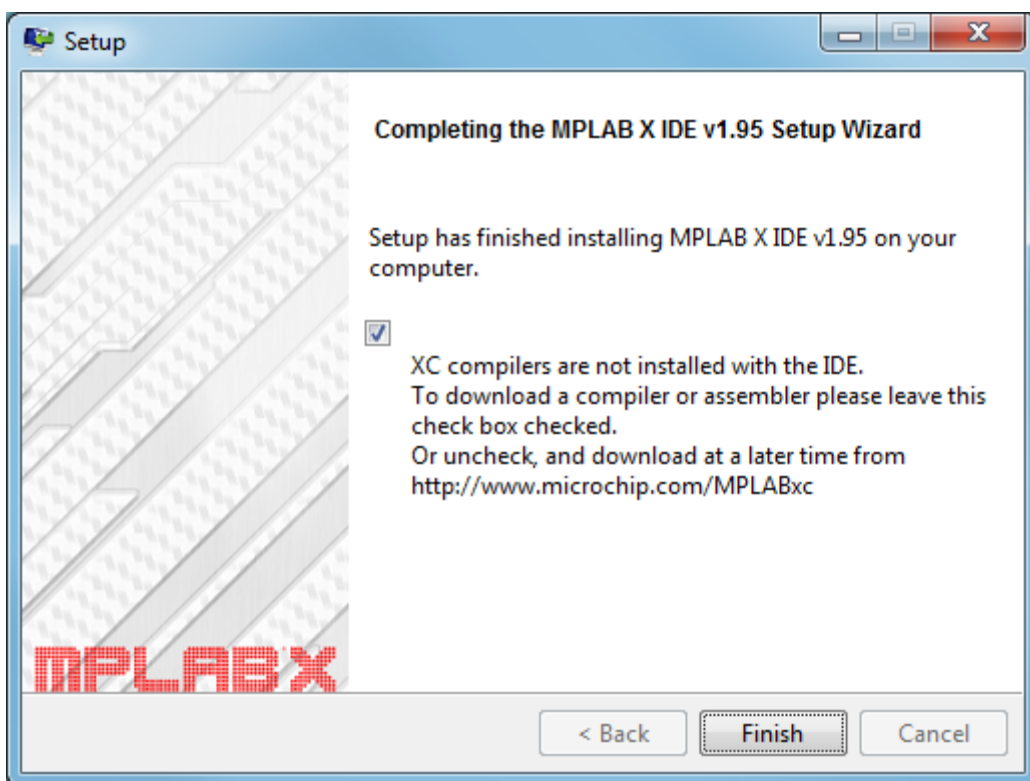
Installation

Before installing XC8, you should install the MPLAB X IDE (if you have not already done so).

Download the MPLAB X IDE installer for your platform (Windows, Linux or Mac) from the MPLAB X download page at www.microchip.com, and then run it.

There are no installation options (other than being able to choose the installation directory). It's an "all or nothing" installer, including the MPASM assembler and support for all of Microchip's PIC MCUs and development tools.

When the installation completes, you are prompted to download and one (or more) of Microchip's "XC" series of C compilers:



Check the box to indicate that you wish to download an XC compiler, then click 'Finish' to finish the MPLAB X installation.

Your browser will now open to the XC compiler download page at www.microchip.com⁷, where you can download the XC8 installer for your platform (Windows, Linux or OS X).

When you run the XC8 installer, after accepting the license agreement, you will be presented with various options such as whether you are installing a network license server (no) or if you want to configure a network client (no), and settings concerning compatibility with the older C18 compiler (we don't need them), but it's ok to accept the defaults and click 'Next' all the way through the installation.

At the end of the installation you are given the opportunity to purchase a license, get an evaluation license, or enter a license key, but you should just click 'Next' to use the compiler in "Free mode".

⁷ if you had already installed MPLAB X, and only want to install XC8 at this time, you will need to go to www.microchip.com and find this page yourself...

Data Types

One of the problems with implementing ANSI-standard C on microcontrollers is that there is often a need to work with individual bits, while the smallest data-type included in the ANSI standard is ‘char’, which is normally considered to be a single byte, or 8 bits. Another problem is the length of a standard integer (‘int’) is not defined, being implementation-dependent. Whether an ‘int’ is 16 or 32 bits is an issue on larger systems, but it makes a much more significant difference to code portability on microcontrollers. Similarly, the sizes of ‘float’, ‘double’, and the effect of the modifiers ‘short’ and ‘long’ is not defined by the standard.

So various compilers use different sizes for the “standard” data types, and for microcontroller implementations it is common to add a single-bit type as well – generally specific to that compiler.

Here are the data types and sizes supported by XC8 and, for comparison, the size of the same data types in CCS PCB⁸:

Type	XC8	CCS PCB
bit	1	-
char	8	8
short	16	1
int	16	8
short long	24	-
long	32	16
float	24 or 32	32
double	24 or 32	-

You’ll see that very few of these line up; the only point of agreement is that ‘char’ is 8 bits!

XC8 defines a single ‘bit’ type, unique to XC8.

The “standard” ‘int’ type is 16 bits in XC8, but 8 bits in CCS PCB.

But by far the greatest difference is in the definition of ‘short’: in XC8, it is a synonym for ‘int’, with ‘short’, ‘int’ and ‘short int’ all being 16-bit quantities, whereas in CCS PCB, ‘short’ is a single-bit type.

Finally, note that ‘double’ floating-point variables in XC8 can be either 24 or 32 bits; this is set by a compiler option. 32 bits may be a higher level of precision than is needed in most applications for small applications, so XC8’s ability to work with 24-bit floating point numbers can be useful.

To make it easier to create portable code, XC8 provides the ‘stdint.h’ header file, which defines the C99 standard types such as ‘uint8_t’ and ‘int16_t’.

To show how to use MPLAB X and XC8 to develop PIC programs, it’s best to work through an example.

Example Circuit

We now have enough background information to design a circuit to light an LED.

We’ll need a regulated power supply, let’s assume 5 V, connected to VDD and VSS. And remember that we should add a bypass capacitor, preferably a 100 nF (or larger) ceramic, across it.

We’ll also need an LED of course, and a resistor to limit the current.

Although the PIC12F1501 can supply up to 25 mA from a single pin, 10 mA is more than enough to adequately light most LEDs. With a 5 V supply and assuming a red or green LED with a forward voltage of around 2 V, the voltage drop across the resistor will be around 3 V.

Applying Ohm’s law, $R = V / I = 3 \text{ V} \div 10 \text{ mA} = 300 \Omega$. Since precision isn’t needed here (we only need “about” 10 mA), it’s ok to choose the next highest “standard” E12 resistor value, which is 330 Ω . It

⁸ a compiler used in the [baseline C](#) tutorial series

means that the LED will draw less than 10 mA, but that's a good thing, because, if we're going to use a PICKit 3 to power the circuit, we need to limit overall current consumption to 30 mA, because that is the maximum current the PICKit 3 can supply.

Finally, we need to connect the LED to one of the PIC's pins.

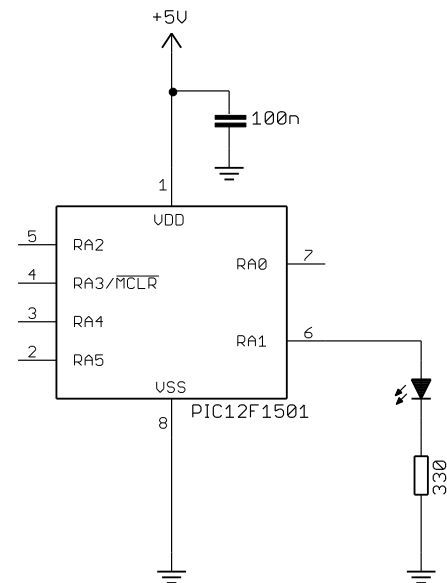
We can't choose RA3, because it's input-only.

If you're using the Gooligum training board, you could choose any of the other pins, but if you use the Microchip LPC Demo Board to implement the circuit, it's not a good idea to use RA0, because it's connected to a trimpot on the LPC demo board, which would divert current from the LED.

So, we'll use RA1, giving the circuit shown on the right.

Simple, isn't it? Modern microcontrollers really do have minimal requirements.

Of course, some connections are also needed for the ICSP (programmer) signals. These will be provided by your development board, unless you are building the circuit yourself. But the circuit as shown here is all that is needed for the PIC to run, and light the LED.



Gooligum training and development board instructions

If you have the Gooligum training board, you can use it to implement this circuit.

Plug the PIC12F1501 into the top section of the 14-pin IC socket – the section marked '12F'⁹.

Connect a shunt across the jumper (JP12) on the LED labelled 'RA1', and ensure that every other jumper is disconnected.

Plug your PICKit 3 programmer into the ICSP connector on the training board, with the arrow on the board aligned with the arrow on the PICKit, and plug the PICKit into a USB port on your PC.

The PICKit 3 can supply enough power for this circuit, so there is no need to connect an external power supply.

Microchip Low Pin Count Demo Board instructions

If you are using Microchip's LPC Demo Board, you'll need to take some additional steps.

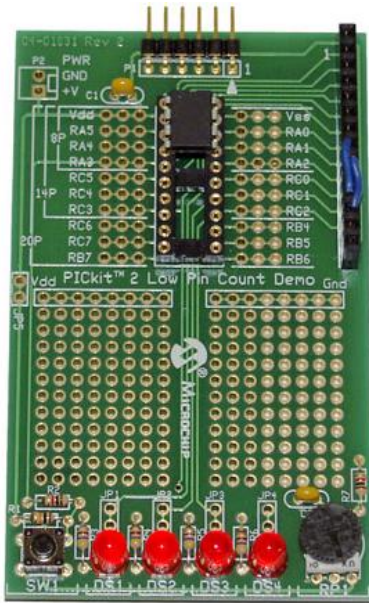
Although the board provides four LEDs, they cannot be used directly with a 12F1501 (or any 8-pin PIC), because those LEDs are connected to DIP socket pins which are only used with 14-pin and 20-pin devices.

However, the circuit can be readily built by adding an LED, a 330 Ω resistor and a piece of wire to the LPC Demo Board, as illustrated on the right.

In the pictured board, a green LED is wired to RA1 and a red LED to RA2; we'll use both LEDs in later lessons. Jumper blocks have been added so that these LEDs can be easily disconnected from the PIC, to facilitate prototyping other circuits. These jumpers are wired in series with each LED.



⁹ Note that, although the PIC12F1501 comes in an 8-pin package, **it will not work** in the 8-pin '10F' socket. You must install it in the '12F' section of the 14-pin socket.



If you prefer not to solder components onto your demo board, you can use the LEDs on the board, labelled ‘DS1’ to ‘DS4’, by making connections on the 14-pin header on the right of the demo board, as shown on the left. This header makes available all the 12F1501’s pins, RA0 – RA5, as well as power (+5 V) and ground. It also brings out the additional pins, labelled ‘RC0’ to ‘RC5’, available on 14-pin PIC devices.

The LEDs are connected to the pins labelled ‘RC0’ to ‘RC3’ on the IC socket, via 470 Ω resistors (and jumpers, if you choose to install them). ‘DS1’ connects to pin ‘RC0’, ‘DS2’ to ‘RC1’, and so on.

So, to connect LED ‘DS2’ to pin RA1, simply connect the pin labelled ‘RA1’ to the pin labelled ‘RC1’, which can be done by plugging a short piece of solid-core hook-up wire between pins 8 and 11 on the 14-pin header.

Similarly, to connect LED ‘DS3’ to pin RA2, simply connect header pins 9 and 12.

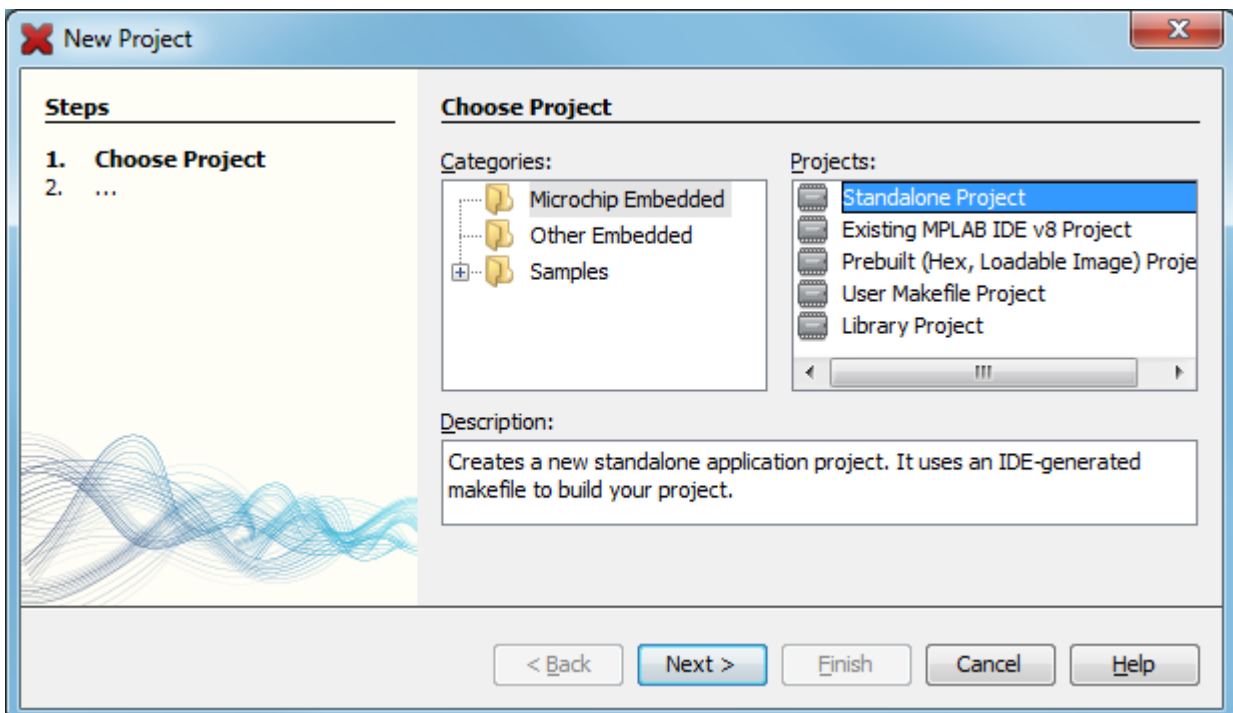
That’s certainly much easier than soldering, so why bother adding LEDs to the demo board? The only real advantage is that, when using 14-pin and 20-pin PICs later, you may find it useful to have LEDs available on RA1 and RA2, while leaving RC0 – RC3 available to use, independently. In any case, it is useful to leave the 14-pin header free for use as an expansion connector, to allow you to build more complex circuits, such as those found in the later tutorial lessons.

Creating a New Project in MPLAB X

When you first run MPLAB X, you will see the “Learn & Discover” tab, on the Start Page.

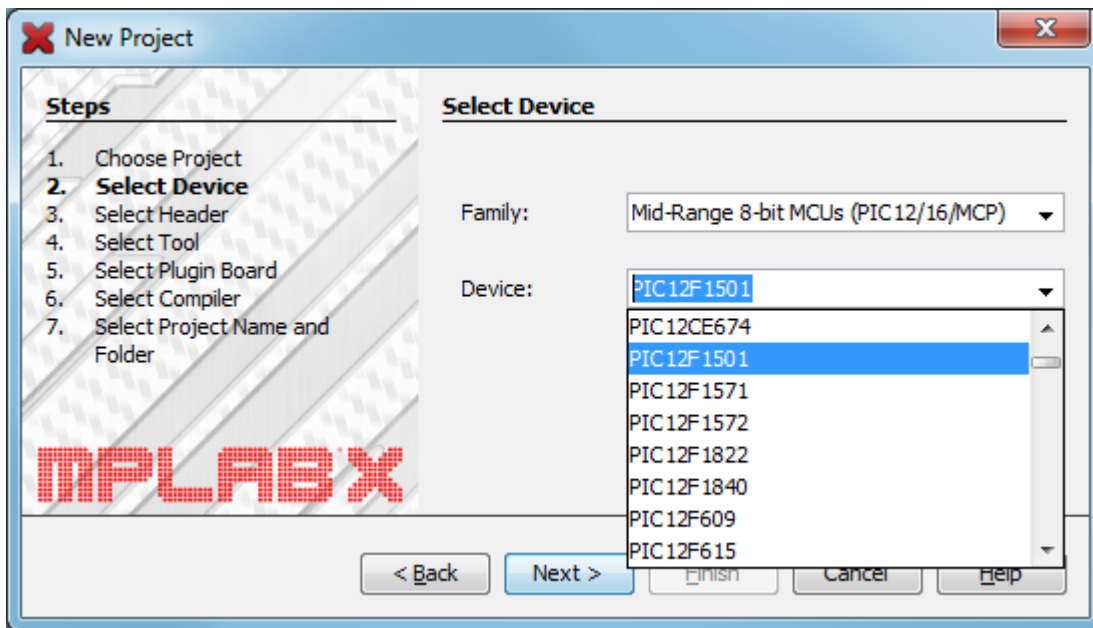
To start a new project, you should run the New Project wizard, by clicking on ‘Create New Project’.

In the first step, you need to specify the project category. Choose ‘Standalone Project’:



Next, select the PIC family and device.

In our case, we need ‘Mid-Range 8-bit MCUs’ as the family, and ‘PIC12F1501’ as the device:

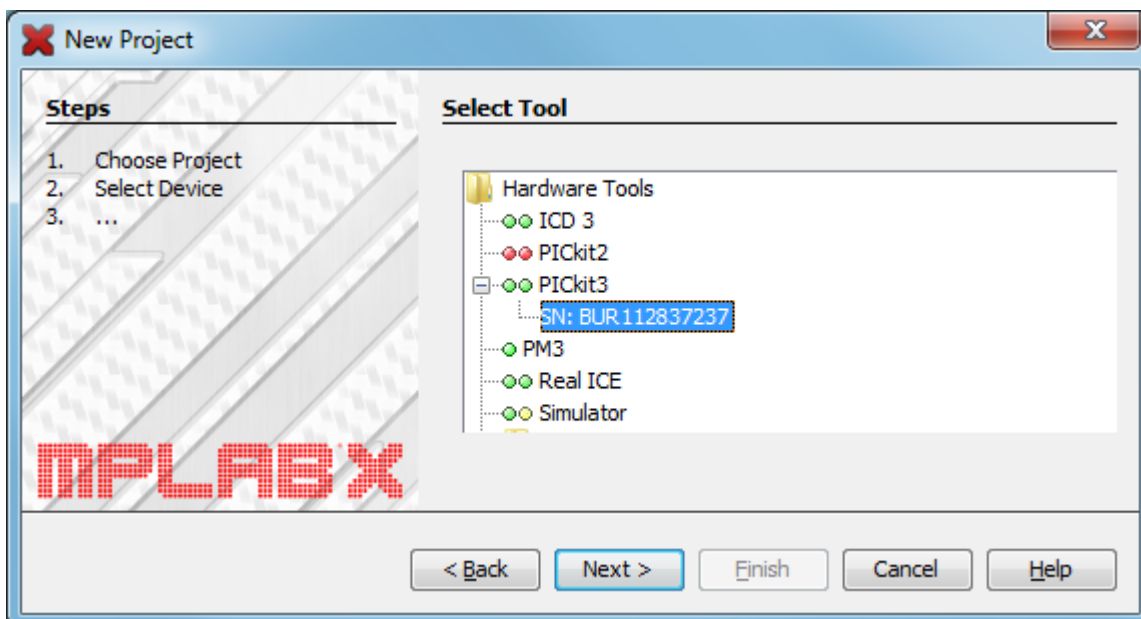


The third step allows you to optionally select a debug header.

This is a device used to facilitate hardware debugging (see explanation in [lesson 0](#)), especially for PICs (such as the 12F1501) which do not include internal hardware to support debugging. If you are just starting out, you are unlikely to have one of these debug headers, and you don't need one for these tutorials. So, you should not select a header. Just click ‘Next’.

The next step is to select the tool you will use to program your PIC.

First, you should plug in the programmer (e.g. PICKit 3) you intend to use. If it is properly connected to your PC, with a functioning device driver¹⁰, it will appear in the list of hardware tools, and you should select it, as shown:

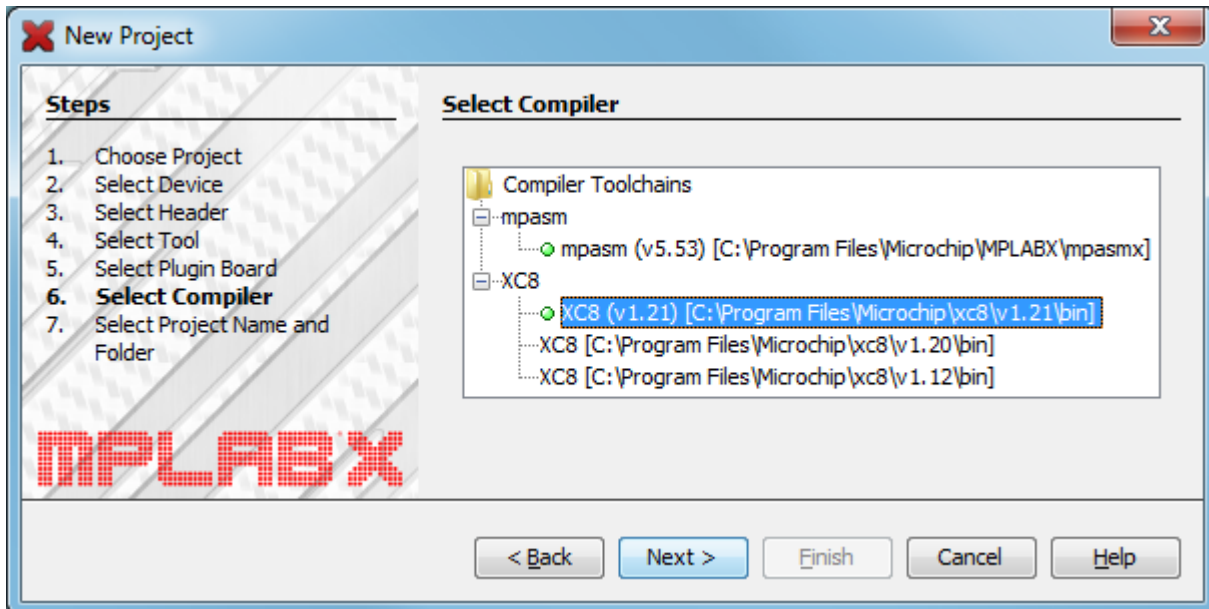


¹⁰ There is no need to install a special device driver for the PICKit 3; it works “out of the box”.

In this case, a PICkit 3 is connected to the PC.

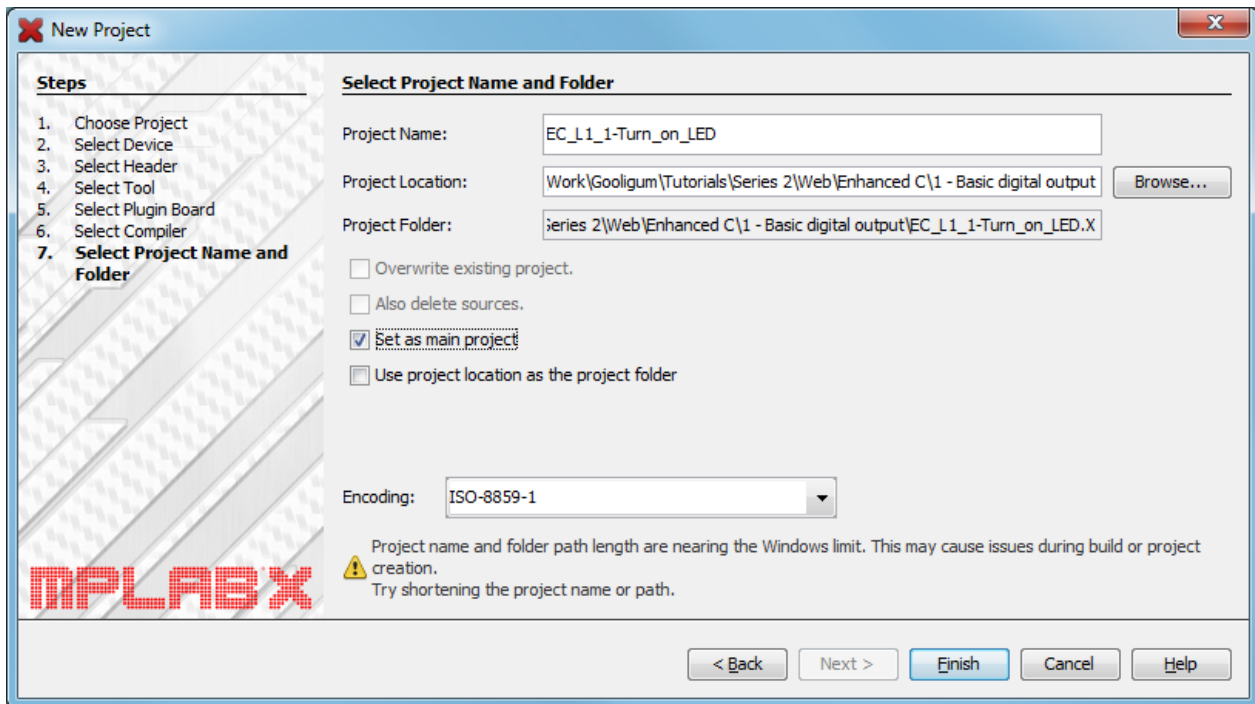
If you have more than one programmer plugged in (including more than one of the same type, such as two PICkit 3s), they will all appear in this list, and you should select the specific one you intend to use for this project – you may need to check the serial number. Of course, you probably only have one programmer, so your selection will be easy.

After selecting the hardware tool, you select the compiler you wish to use:



Select “XC8” (taking care to select the version you wish to use, if you have more than one XC8 compiler installed) to specify that this is an XC8 project.

Finally, you need to specify your project's location, and give it a name:



For example, in the environment used to develop these tutorials, all the files related to this lesson, including schematics and documentation, are placed in a folder named '1 – Basic digital output', which is the "Project Location" shown above.

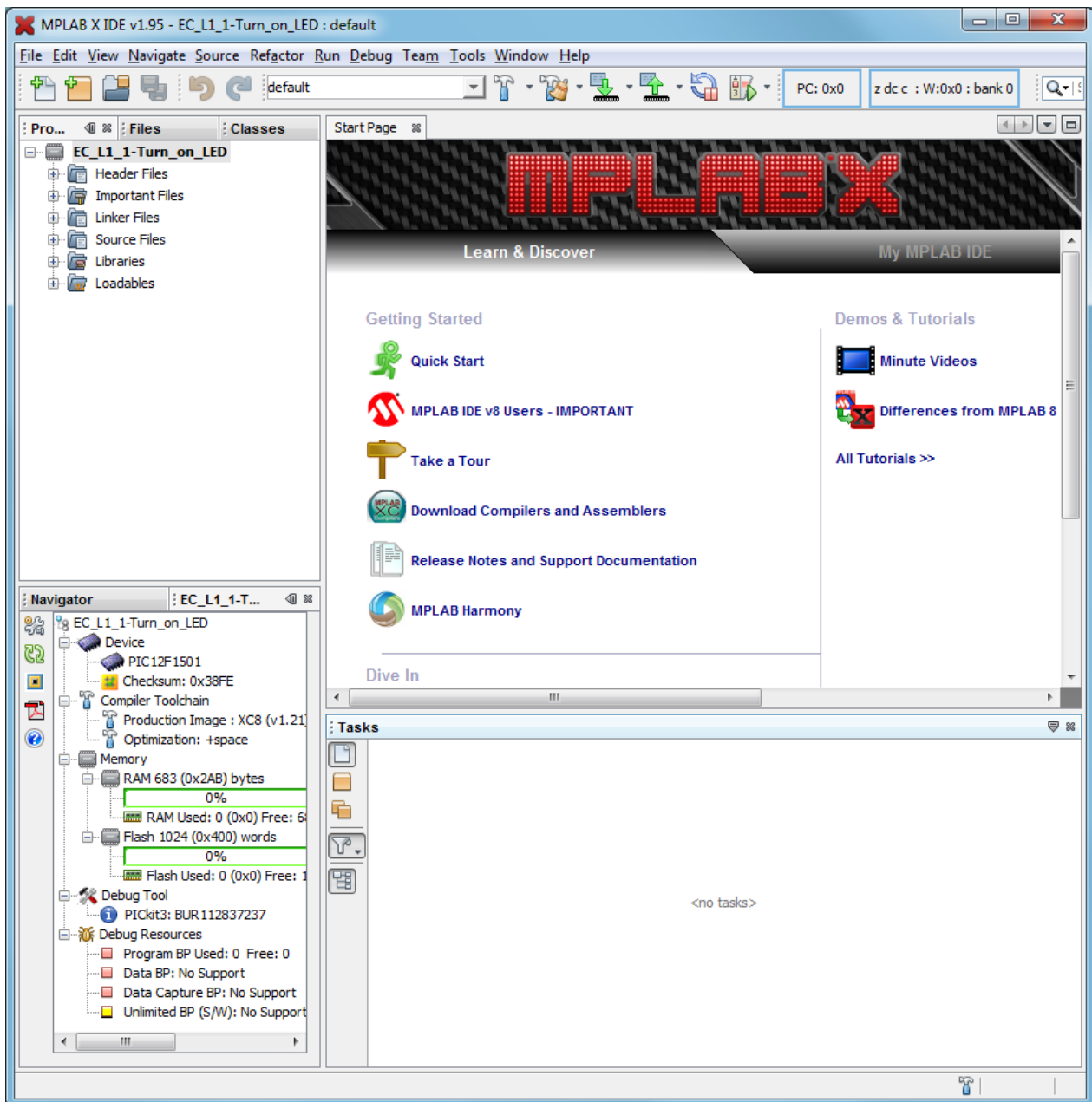
By default, MPLAB X then creates a separate folder for the PIC source code and other files related to this project, in a subfolder that has the same name as the project, with a '.X' on the end. If you wish, you can remove the '.X' extension from the project folder, before you click on 'Finish'.

If you select "Use project location as the project folder", this behaviour changes – the project files are then placed in the "Project Location" folder, instead of being in a separate folder. This isn't recommended, which is why that box is left unchecked above. But if you prefer not to have a separate folder for the MPLAB files, you can select this option.

Note the warning about project name and folder path length. To avoid possible problems, it's best to use shorter names and paths, when using Windows, although in this case it's actually ok.

Since this is the only project we're working on, it doesn't make much difference whether you select "Set as main project"; this is something that is more useful when you are working with multiple projects.

After you click “Finish”, your workspace should look something like this:



The panel in the top left allows you to see and access the various files that comprise your project – we’ll add a source code file (which will appear under “Source Files” in the next step).

The panel in the bottom left shows your project’s configuration and status, such as which device you’re using (12F1501), the selected programmer (shown here as “Debug Tool”) and the amount of PIC program (“Flash”) and data (“RAM”) memory our program is using – 0% for now, because we haven’t created a program yet!

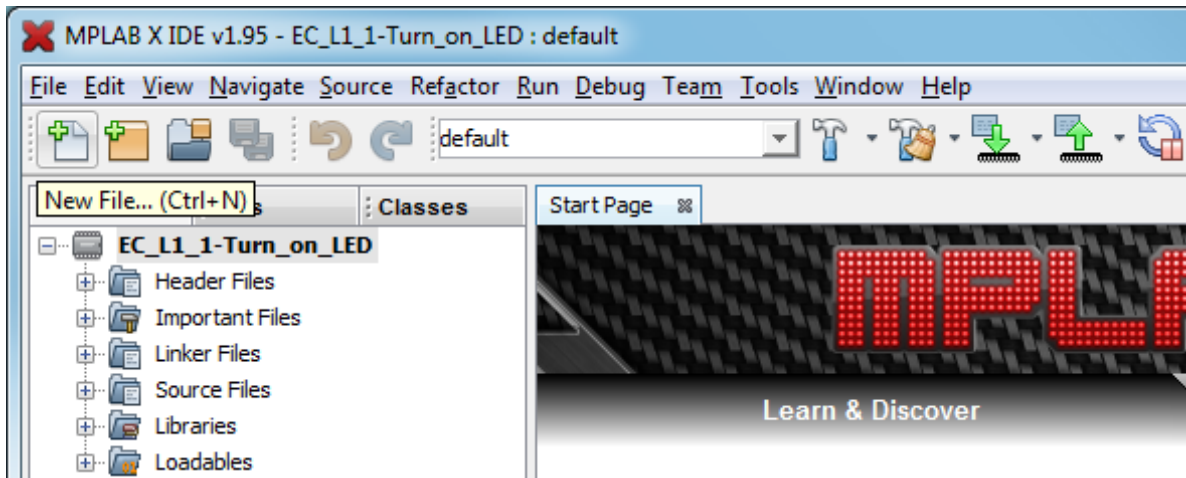
The largest panel, in the upper right, is where you edit your source code.

Below it is a panel where you’ll see the status of processes such as compiling your program and programming the PIC.

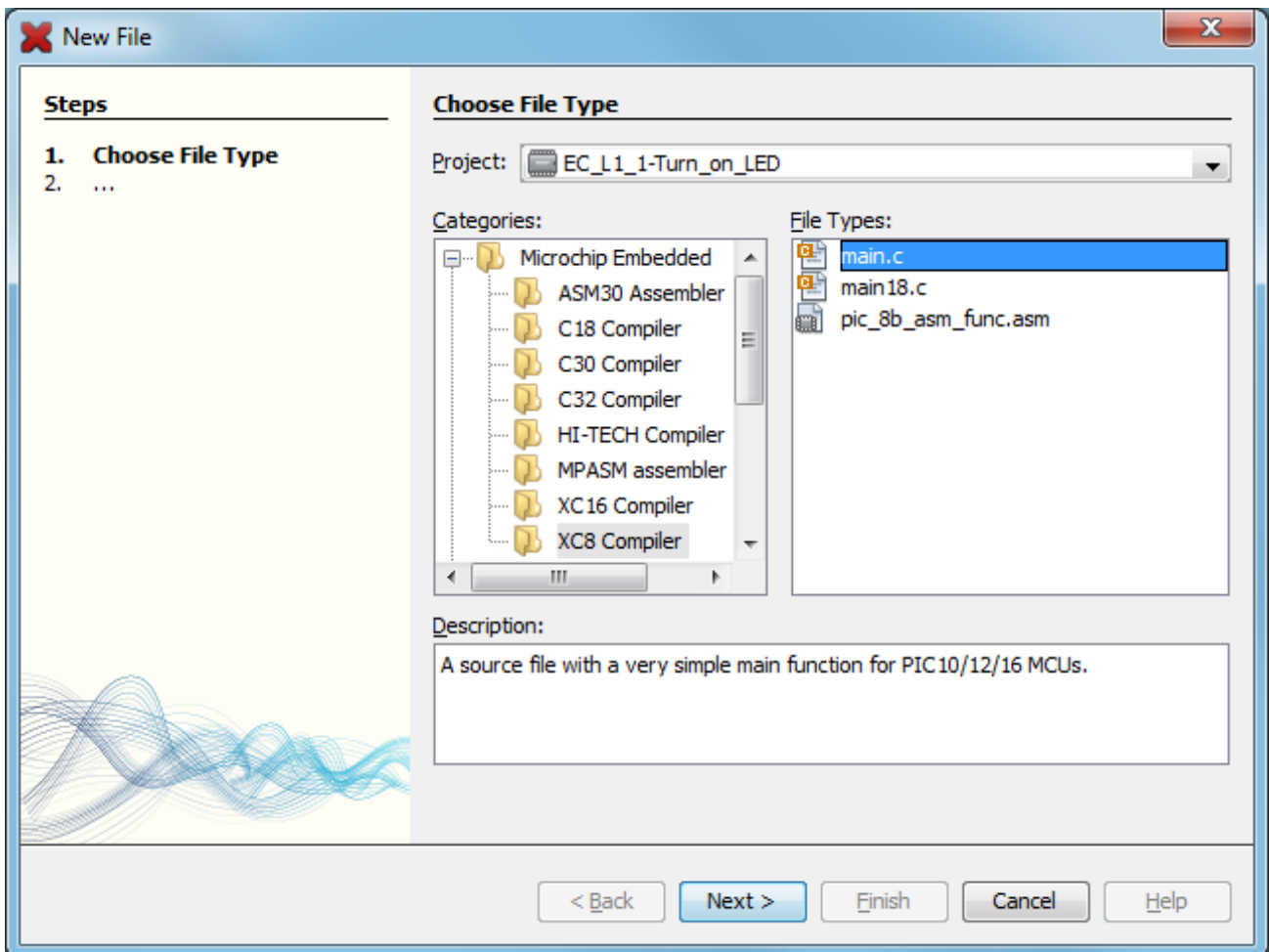
Note that MPLAB X has many, many features that we won’t be exploring in these tutorials. These lessons are about PIC programming, not using MPLAB X. So it’s worth taking some time to explore the training resources available from the “Learn & Discover” tab shown above.

There are a couple of ways to create a new source file and add it to the project.

You could select the “File → New File...” menu item, press Ctrl+N, or click on the “New File” button in the toolbar:



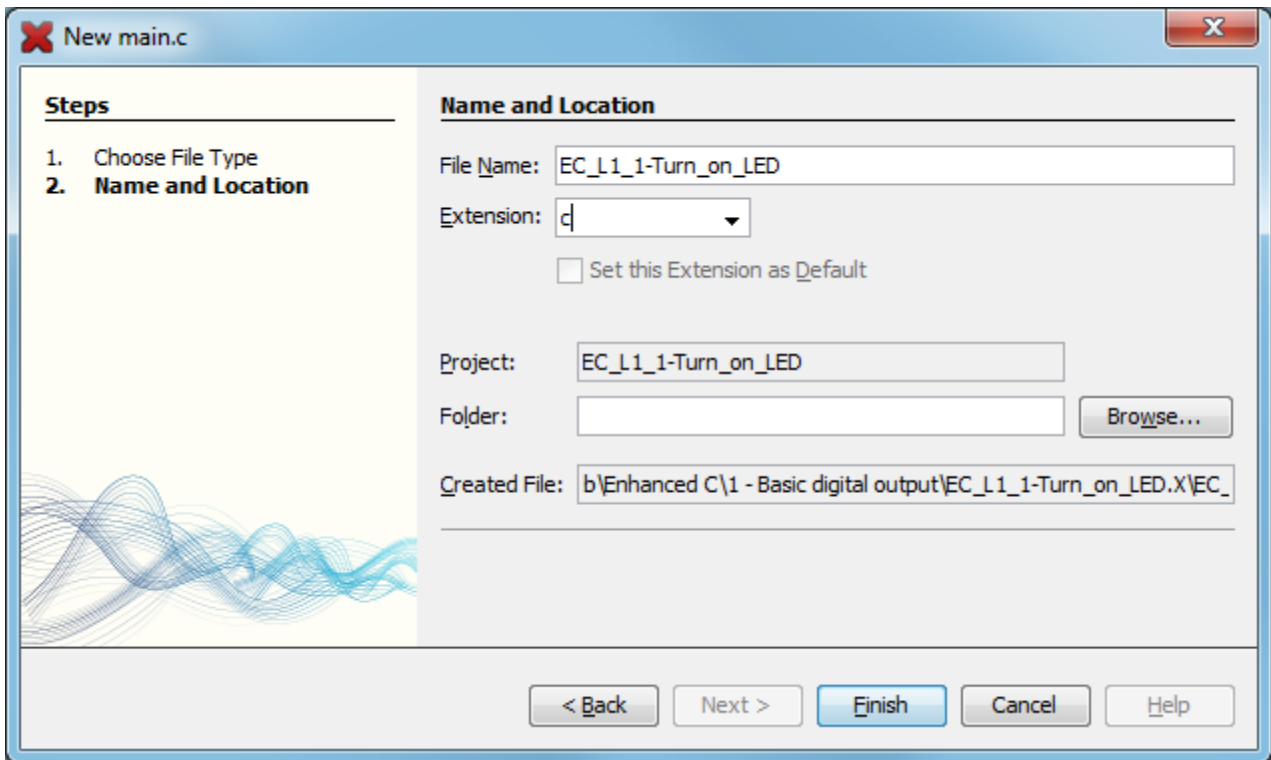
This will open the New File window:



Microchip provide a number of templates to base your source file on.

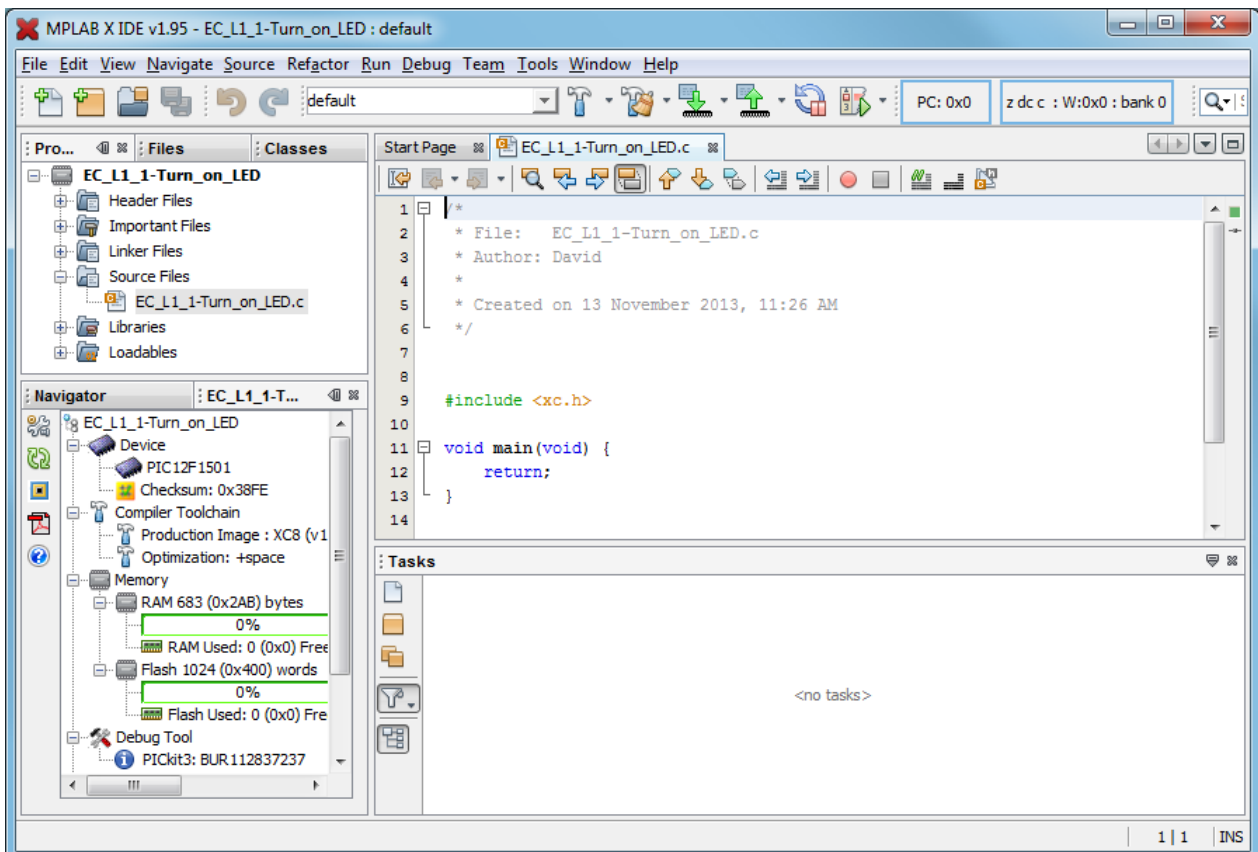
We’re creating an XC8 compiler project, so navigate to “XC8 compiler”, within the “Microchip Embedded” category, then select “main.c”, as shown.

After you click ‘Next’, you have the option of naming your file:



The “Folder” field allows you to place your file in a different directory, not necessarily within the project folder. You wouldn’t normally do that, so it’s ok to leave it blank, as shown here.

Your source file should now appear under ‘Source Files’ in the project tree, and you should be able to see the source code in the editor window, as shown:



If you don't see the source code, you may need to click on the tab at the top of the editor pane, or double-click the source file name in the project tree.

Now you can finally start working on your code!

XC8 Source Code

A large program can consist of a number of source files, each containing various modules or definitions, or, in a simple example such as this one, you may have only a single source file.

Regardless of whether a source file stands on its own or is part of a larger program, it is usual to begin it with a block of comments, providing essential information about the source file such as what it's called, the last modification date and current version (and sometimes a history of previous versions, what has changed in this version, and who changed it), who wrote it, and a general description of what the program or module does.

It can also be useful to include a "Files required" section. This is helpful in larger projects, where your code may rely on other files or modules; you can list any dependencies here.

It is also a good idea to include information on what processor this code is written for; useful if you move it to a different PIC later. You should also document what each pin is used for. It's common, when working on a project, to change the pin assignments – often to simplify the circuit layout. Clearly documenting the pin assignments helps to avoid making mistakes when they are changed! And when writing in C, it is a good idea to state which compiler has been used because, as we have seen for data types, C code for microcontrollers is not necessarily easily portable.

So we might use something like:

```

/*****
*
*   Filename:      EC_L1_1-Turn_on_LED.c
*   Date:         13/11/13
*   File Version:  0.1
*
*   Author:       David Meiklejohn
*   Company:      Gooligum Electronics
*
*****
*
*   Architecture: Enhanced Mid-range PIC
*   Processor:    12F1501
*   Compiler:     MPLAB XC8 v1.21 (Free mode)
*
*****
*
*   Files required: none
*
*****
*
*   Description:   Lesson 1, example 1
*
*   Turns on LED. LED remains on until power is removed.
*
*****
*
*   Pin assignments:
*       RA1 = indicator LED
*
*****/

```

Note that the file version is ‘0.1’. I don’t call anything ‘version 1.0’ until it works; when I first start development I use ‘0.1’. You can use whatever scheme makes sense to you, as long as you’re consistent.

You can type these comments into the start of the source code in the editor pane – or, better, copy and paste them from the source file provided with this lesson.

Most of the symbols relevant to specific processors, which allow us to access registers such as `PORTA`, are defined in *header files*. In XC8 this is done by including a single “catch-all” file: “`xc.h`”. This file identifies the processor being used, and then includes processor-specific header files as appropriate.

So our next line, which is already in place in the template file and should be at the start of every XC8 program, is:

```
#include <xc.h>
```

Next, we need to configure the processor.

The 12F1501 has a number of options that are selected by setting various bits in a pair of “configuration words”, sometimes known as “fuses”, which sit outside the normal address space.

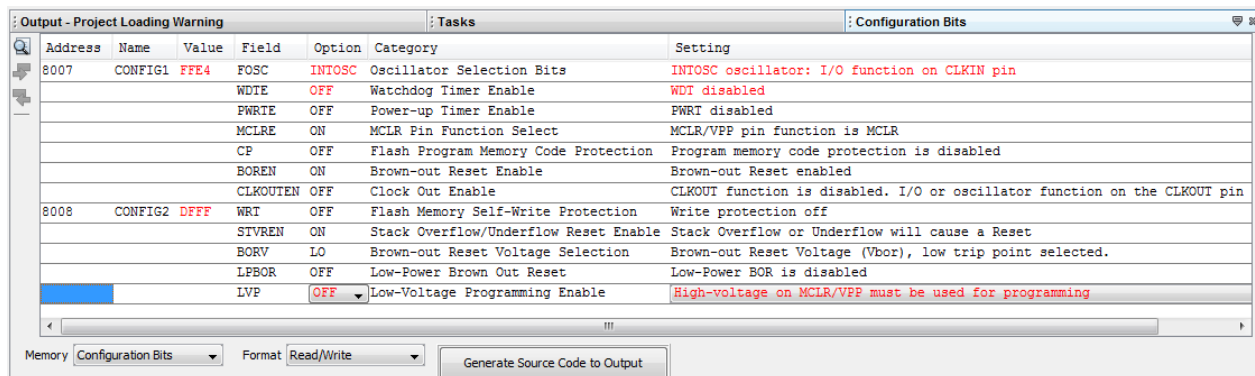
Configuration pragmas (“`#pragma config`” directives) are used to specify these configuration bits.

We could look up the configuration bits in the data sheet and type in the appropriate configuration pragmas ourselves – and in fact, when you’re creating a new program, based on one that you’ve worked on before (as you’ll often do), it’s quite normal to directly edit the `#pragma config` directives.

However, MPLAB X includes a generator which can create these directives for us.

To use it, select the “Window → PIC Memory Views → Configuration Bits” menu item.

You will see the processor configuration options in the “Configuration Bits” window under the editor pane:



As you can see, there are quite a few options, but you only need to change three (shown in red, above):

FOSC = INTOSC

WDTE = OFF

LVP = OFF

When you have made these selections, click on the ‘Generate Source Code to Output’ button.

The generated source code will appear in the “Config Bits Source” tab in the “Output” window:

```

// PIC12F1501 Configuration Bit Settings

#include <xc.h>

// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.

// CONFIG1
#pragma config FOSC = INTOSC // Oscillator Selection Bits (INTOSC oscillator: I/O function on CLKIN pin)
#pragma config WDTE = OFF // Watchdog Timer Enable (WDT disabled)
#pragma config PWRTE = OFF // Power-up Timer Enable (PWRT disabled)
#pragma config MCLRE = ON // MCLR Pin Function Select (MCLR/VPP pin function is MCLR)
#pragma config CP = OFF // Flash Program Memory Code Protection (Program memory code protection is disabled)
#pragma config BOREN = ON // Brown-out Reset Enable (Brown-out Reset enabled)
#pragma config CLKOUTEN = OFF // Clock Out Enable (CLKOUT function is disabled. I/O or oscillator function on the CLKOUT pin)

// CONFIG2
#pragma config WRT = OFF // Flash Memory Self-Write Protection (Write protection off)
#pragma config STVREN = ON // Stack Overflow/Underflow Reset Enable (Stack Overflow or Underflow will cause a Reset)
#pragma config BORV = LO // Brown-out Reset Voltage Selection (Brown-out Reset Voltage (Vbor), low trip point selected.)
#pragma config LPBOR = OFF // Low-Power Brown Out Reset (Low-Power BOR is disabled)
#pragma config LVP = OFF // Low-Voltage Programming Enable (High-voltage on MCLR/VPP must be used for programming)

```

You can then copy and paste (using the usual select and right-click method) the `#pragma` directives into your source code in the editor window, immediately after the `#include <xc.h>` directive – which, as you can see is also part of this configuration bits output – you don’t need to include it twice!

We’ll examine these in greater detail in later lessons, but briefly the options being set here are:

- `FOSC = INTOSC`

This selects the internal RC oscillator as the clock source.

Every processor needs a clock – a regular source of cycles, used to trigger processor operations such as fetching the next program instruction.

Most modern PICs, including the 12F1501, include an internal ‘RC’ oscillator, which can be used as the simplest possible clock source, since it’s all on the chip! It’s built from passive components – resistors and capacitors – hence the name RC.

The internal RC oscillator on the 12F1501 runs at approximately 16 MHz and by default this is divided down to 500 kHz. Program instructions are processed at one quarter this speed: 125 kHz, or 8 μ s per instruction.

Alternatively, the 12F1501 can use a (possibly more accurate) external clock signal, via the CLKIN pin. This shares its physical pin with RA5, so if you’re using an external clock, you can’t use the RA5 pin for I/O.

To turn on an LED, we don’t need accurate timing, so we’ll use the internal RC oscillator.

- `WDTE = OFF`

Disables the watchdog timer.

This is a way of automatically restarting a crashed program; if the program is running properly, it continually resets the watchdog timer. If the timer is allowed to expire, the program isn’t doing what it should, so the chip is reset and the crashed program restarted – see lesson 5.

The watchdog timer is very useful in production systems, but a nuisance when prototyping, so we’ll leave it disabled.

- `PWRTE = OFF`

Disables the power-up timer.

When a power supply is first turned on, it can take a while for the supply voltage to stabilise, during which time the PIC's operation may be unreliable. If the power-up timer is enabled, the PIC is held in reset (it does not begin running the user program) for some time, nominally 64 ms, after the supply voltage reaches a minimum level.

However, since we have enabled the brown-out reset facility, which will prevent the device from starting until the supply voltage is high enough, we don't need to also enable the power-up timer.

- `MCLRE = ON`

Enables external reset, or "master clear" ($\overline{\text{MCLR}}$) on pin 4.

If external reset is enabled, pulling this pin low will reset the processor. Or, if external reset is disabled, the pin can be used as an input: RA3. That's why, on the circuit diagram, pin 4 is labelled " $\text{RA3}/\overline{\text{MCLR}}$ "; it can be either an input pin or an external reset, depending on the setting of this configuration bit.

The Gooligum training board includes a pushbutton which will pull pin 4 low when pressed, resetting the PIC if external reset is enabled. The PICkit 3 is also able to pull the reset line low, allowing MPLAB to control $\overline{\text{MCLR}}$ (if enabled) – useful for starting and stopping your program.

So unless you need to use every pin for I/O, it's a good idea to enable external reset.

- `CP = OFF`

Turns off code protection.

When your code is in production and you're selling PIC-based products, you may not want competitors stealing your code. If you specify "`CP = ON`" instead, your code will be protected, meaning that if someone tries to use a PIC programmer to read it, all they will see are zeros.

Since we're not designing anything for sale, we'll make our lives easier by leaving code protection turned off.

- `BOREN = ON`

Enables brown-out resets.

The PIC's operation can become unreliable if the supply voltage drops too low, which can happen during a *brown-out*, when the supply voltage sags, but does not fall quickly to zero. The 12F1501 has brown-out detect circuitry, which will reset the PIC in a brown-out situation, if brown-out resets are enabled.

Although your power supply is not likely to suffer from brown-outs, it doesn't hurt to have this option enabled – just in case.

- `CLKOUTEN = OFF`

Regardless of whether the internal RC oscillator or an external clock signal is used as the processor clock (FOSC) source, the instruction clock (FOSC/4) can optionally be output on the CLKOUT pin, to allow other devices to be synchronised with the PIC's operation. CLKOUT shares its pin with RA4, so if you're using the clock out facility, you can't use RA4 for I/O.

We don't need to use CLKOUT, so we will leave this feature disabled.

- `WRT = OFF`

Disables flash memory write protection.

Many newer PIC devices, including the 12F1501, are capable of writing to their flash (program) memory. This is useful in a number of situations, including boot loaders, which allow program firmware to be updated easily in the field, or to store data long term (flash memory being non-volatile).

Of course, you don't want your program to be overwritten by mistake! To prevent that from happening, you may wish to write-protect all or some of the flash memory.

Nevertheless, it's safe in this example to leave flash write protection disabled.

- `STVREN = ON`
Enables stack overflow/underflow resets.

The *stack* is a special set of registers which are usually used by the C compiler when calling functions. Its use is managed by the C compiler and should not be something that, as a C programmer, you usually have to be concerned about.

A stack overflow or underflow should never happen – but if it does, it means that something has gone wrong, and your program probably isn't working properly. If this option is enabled, the PIC will reset itself if a stack overflow or underflow occurs – hopefully allowing your program to recover. So although, like brown-out resets, this type of event “shouldn't happen”, it doesn't hurt to leave this feature enabled.

- `BORV = LO`
Selects the low brown-out reset voltage option

This option selects the voltage level at which the brown-out reset (if enabled) will be tripped.

- `LPBOR = OFF`
Disables low-power brown-out resets.

The 12F1501 also has a lower-power brown-out reset facility; we can leave it disabled.

- `LVP = OFF`
Disables low-voltage programming.

Normally, to program the device, a high voltage (around 12 V) must be applied to the VPP pin. Low-voltage programming mode avoids the need for this high voltage, but we don't need it because the PICkit 3 can operate in the traditional high-voltage programming mode.

The comments in the generated configuration code are very long, but a little confusing: does “Watchdog Timer Enable (WDT disabled)” mean that the watchdog timer is enabled or not?

So we'll modify the comments and rearrange the configuration code, grouping related options (note that it's ok to have more than one configuration pragma in each line) to make it more informative:

```

/***** CONFIGURATION *****/
// ext reset, internal oscillator (no clock out), no watchdog timer
#pragma config MCLRE = ON, FOSC = INTOSC, CLKOUTEN = OFF, WDTE = OFF
// brownout resets enabled, low brownout voltage, no low-power brownout reset
#pragma config BOREN = ON, BORV = LO, LPBOR = OFF
// no power-up timer, no code protect, no write protection
#pragma config PWRTE = OFF, CP = OFF, WRT = OFF
// stack resets on, high-voltage programming
#pragma config STVREN = ON, LVP = OFF

```

Finally, in the template code, we have:

```

void main(void) {
    return;
}

```

As with most C compilers, the entry point for “user” code is a function called ‘main()’.

In other words, this is where our program starts.

Declaring `main()` as “`void main(void)`” like this is “correct”, but the use of “`void`” isn’t strictly necessary. This declaration is saying that the program does not receive or return any values – but that is only relevant when the program is being run by an operating system.

Programs running on small microcontrollers, such as enhanced mid-range PICs, have nowhere to go if they “finish” – it’s not like programs running on a PC with an operating system. If a PIC program is allowed to run past its “end”, it will attempt to execute whichever “instructions” happen to be in the (uninitialised, non-programmed) remainder of the program memory. Whatever the “program” is doing at that point, it’s not under your control and not behaviour that you want.

Using a “return” statement at the end of the program, as in the template code, doesn’t help because, again, there is no operating system for the program to return control to.

So, programs running on small microcontrollers, where there is no operating system, are almost always designed to never finish running. Instead, the usual structure is to have some initialisation code which runs when the program starts, often some interrupt services routines which will be run when interrupts are triggered, and a *main loop*, which repeats the same processes “forever” – that is, until the power is cut off or the device is reset.

How you declare `main()` is really a question of personal style, but our C programs will normally be structured as:

```
void main()
{
    // configuration and initialisation code goes here

    for (;;)
    {
        // main loop code
        ;
    }
}
```

This is where we place the code to turn on the LED!

Turning on the LED

To turn on the LED on RA1, we need to do two things:

- Configure RA1 as an output
- Set RA1 to output a high voltage

We could leave the other pins configured as inputs, or set them to output a low. Since, in this circuit, they are not connected to anything, it doesn’t really matter. But for the sake of this exercise, we’ll configure them as inputs.

When an enhanced mid-range PIC is powered on, all pins are configured by default as inputs.

To configure only RA1 as an output, we have to clear bit 1 of the TRISA register, leaving all the other bits in TRISA set.

XC8 makes the PIC’s special function registers available as variables defined in the header files.

Loading the TRISA register with 111101b (clearing bit 1, configuring RA1 as an output) is simply:

```
TRISA = 0b111101;    // configure RA1 (only) as an output
```

Alternatively, to make it clear that we're clearing bit 1, we can use bitwise operators in a logical expression:

```
TRISA = ~(1<<1);           // configure RA1 (only) as an output
```

Of course, whether that form does seem clearer will depend on how familiar you are with C logical expressions!

To make RA1 output a 'high', we have to set bit 1 of PORTA to '1', which we could do with:

```
PORTA = 0b000010;        // set RA1 high
```

Although there is no risk of running into read-modify-write problems when updating the port register in a single write operation like this, to avoid potential problems in other situations it is better to get into the habit of only ever writing to LATA to modify output pins:

```
LATA = 0b000010;        // set RA1 high
```

Or, if you prefer:

```
LATA = 1<<1;           // set RA1 high
```

Either way, we're clearing every bit in the LATA register, while setting LATA<1> (or LATA1) bit.

That's ok in this example, where we only have a single LED, but normally you'd want to be able to set a single bit while leaving the rest of the register unchanged.

Because this is such a common requirement, XC8 provides a mechanism to allow individual bits, such as LATA1, to be accessed through bit-fields defined in the header files.

For example, the "pic12f1501.h" header file defines a union called LATABits, containing a structure with bit-field members LATA0, LATA1, etc.

So, to set RA1 to '1', we can write:

```
LATABits.LATA1 = 1;      // set RA1 high
```

Finally, as explained earlier, if we leave it there, when the program gets to the end of this code, it will attempt to execute whatever happens to be in the remainder of the program memory. We need to get the PIC to just sit doing nothing, indefinitely, with the LED still turned on, until it is powered off – which means finishing with an endless loop:

```
for (;;)
{
    // loop forever
    ;
}
```

Once again, this little program has a structure common to most PIC programs: an initialisation section, where the I/O pins and other facilities are configured and initialised, followed by a "main loop", which repeats forever. Although we'll add to it in future lessons, we'll always keep this basic structure of initialisation code followed by a main loop.

Complete program

Putting together all the above, and adding a few more comments, here's the complete C source code for turning on an LED, for the PIC12F1501:

```

/*****
*
*   Description:      Lesson 1, example 1
*
*   Turns on LED.   LED remains on until power is removed.
*
*****
*
*   Pin assignments:
*       RA1 = indicator LED
*
*****/

#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, internal oscillator (no clock out), no watchdog timer
#pragma config MCLRE = ON, FOSC = INTOSC, CLKOUTEN = OFF, WDTE = OFF
// brownout resets enabled, low brownout voltage, no low-power brownout reset
#pragma config BOREN = ON, BORV = LO, LPBOR = OFF
// no power-up timer, no code protect, no write protection
#pragma config PWRTE = OFF, CP = OFF, WRT = OFF
// stack resets on, high-voltage programming
#pragma config STVREN = ON, LVP = OFF

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    TRISA = ~(1<<1);        // configure RA1 (only) as an output

    LATAbits.LATA1 = 1;     // set RA1 high

    /*** Main loop
    for (;;)
    {
        // loop forever
        ;
    }
}

```

That's it! Not a lot of code, really...

Building the Application and Programming the PIC

Now that we have the complete XC8 source, we can build the final application code and program it into the PIC.

This is done in two steps:

- Build the project
- Use a programmer to load the program code into the PIC

The first step, building the project, involves compiling the source files to create object files, and linking these object files, to build the executable code. Normally this is transparent; MPLAB does all of this for you in a single operation. The fact that, behind the scenes, there are multiple steps only becomes important when you start working with projects that consist of multiple source files or libraries of pre-compiled routines.

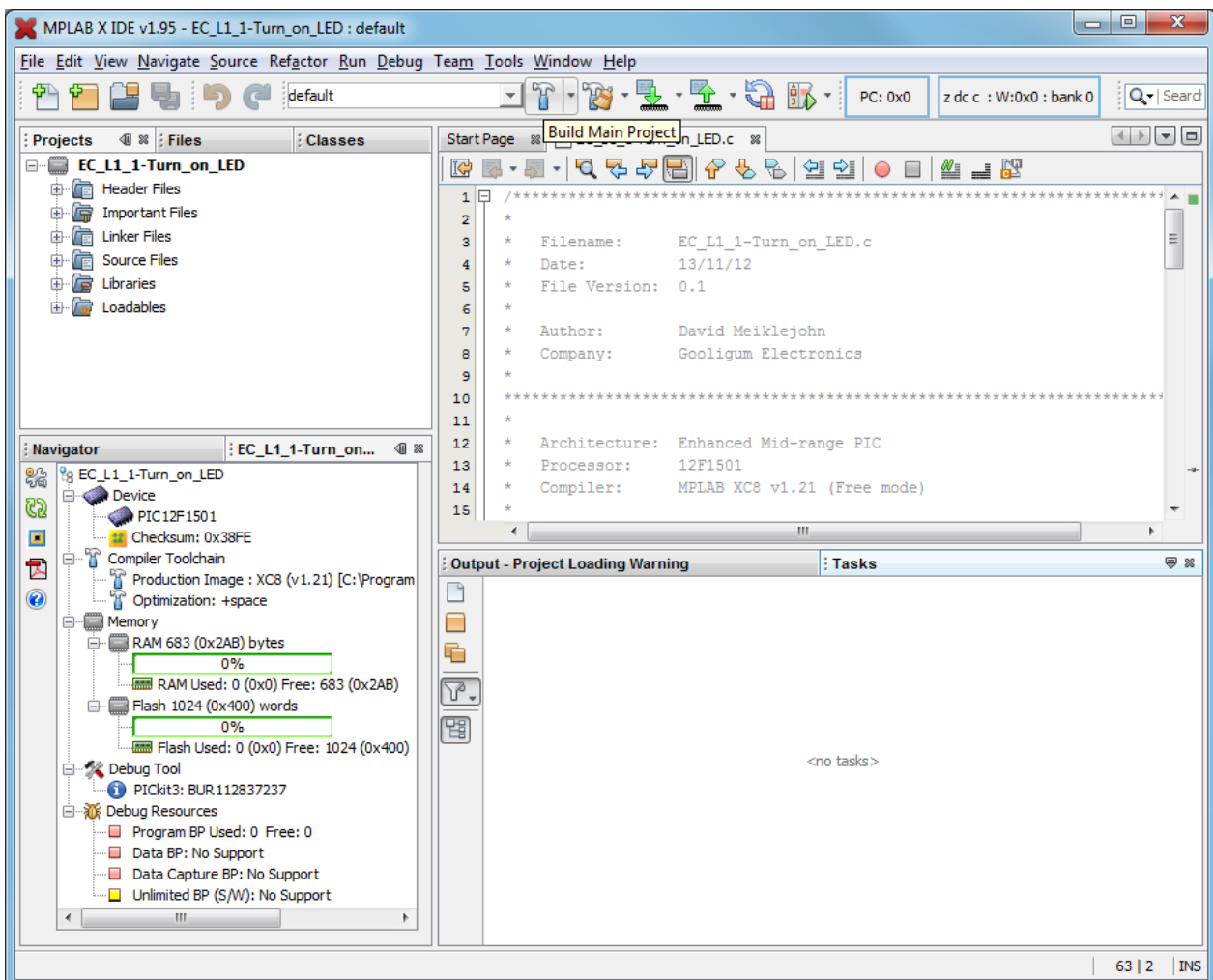
A PIC programmer, such as the PICkit 3, is then used to upload the executable code into the PIC. Although a separate application is sometimes used for this “programming” process, it’s convenient when developing code to do the programming step from within MPLAB, which is what we’ll look at here.

Building the project

Before you build your project using MPLAB X, you should first ensure that it is the “main” project. It should be highlighted in bold in the Projects window.

To set the project you want to work on (and build) as the main project, you should right-click it and select “Set as Main Project”. If you happen to have more than one project in your project window, you can by removing any project you are not actively working on (to reduce the chance of confusion) from the Projects window, by right-clicking it and selecting “Close”.

To build the project, right-click it in the Projects window and select “Build”, or select the “Run → Build Main Project” menu item, or simply click on the “Build Main Project” button (looks like a hammer) in the toolbar:



This will compile any source files which have changed since the project was last built, and link them.

An alternative is “Clean and Build”, which removes any compiled (object) files and then re-compiles all files, regardless of whether they have been changed. This action is available by right-clicking the project in the Projects window, or under the “Run” menu, or by clicking on the “Clean and Build Main Project” button (looks like a hammer with a brush) in the toolbar.

When you build the project, you’ll see messages in the Output window, showing your source files being compiled and linked. Toward the end, you should see:

```
BUILD SUCCESSFUL (total time: 13s)
```

(of course, your total time will probably be different...)

If, instead, you see an error message, you’ll need to check your code and your project configuration.

Programming the PIC

The final step is to upload the executable code into the PIC.

First, ensure that you have connected your PICkit 3 programmer to your Gooligum training board or Microchip LPC Demo Board, with the PIC correctly installed in the appropriate IC socket¹¹, and that the programmer is plugged into your PC.

If you have been following this lesson, you will have specified the programmer when you created your project (in step 4 of the wizard).

The project dashboard, in the panel in the bottom right of the workspace, shows the currently-selected programmer under “Debug Tool”. If you want to change this tool selection, you can right-click your project in the Projects window and select “Properties”, or simply click on the “Project Properties” button on the left side of the project dashboard, as shown:

The screenshot displays the MPLAB IDE interface. On the left, the 'Project Properties' window is open for the project 'EC_L1_1-Turn_on_LED'. It shows the following details:

- Device: 12F1501
- Checksum: 0x8B59
- Compiler Toolchain: Production Image : XC8 (v1.21) [C:\Program Files\Micro...
- Optimization: +space
- Memory:
 - RAM 683 (0x2AB) bytes: 0% used
 - RAM Used: 2 (0x2) Free: 681 (0x2A9)
 - Flash 1024 (0x400) words: 1% used
 - Flash Used: 12 (0xC) Free: 1012 (0x3F4)
- Debug Tool: PICkit3: BUR112837237
- Debug Resources:
 - Program BP Used: 0 Free: 0
 - Data BP: No Support
 - Data Capture BP: No Support
 - Unlimited BP (S/W): No Support

On the right, the 'Output' window shows the following text:

```

Project Loading Warning x EC_L1_1-T
EEPROM space      None
Configuration bits used
ID Location space used

Running this compiler in PRO n
produces code which is typical
See http://microchip.com for n

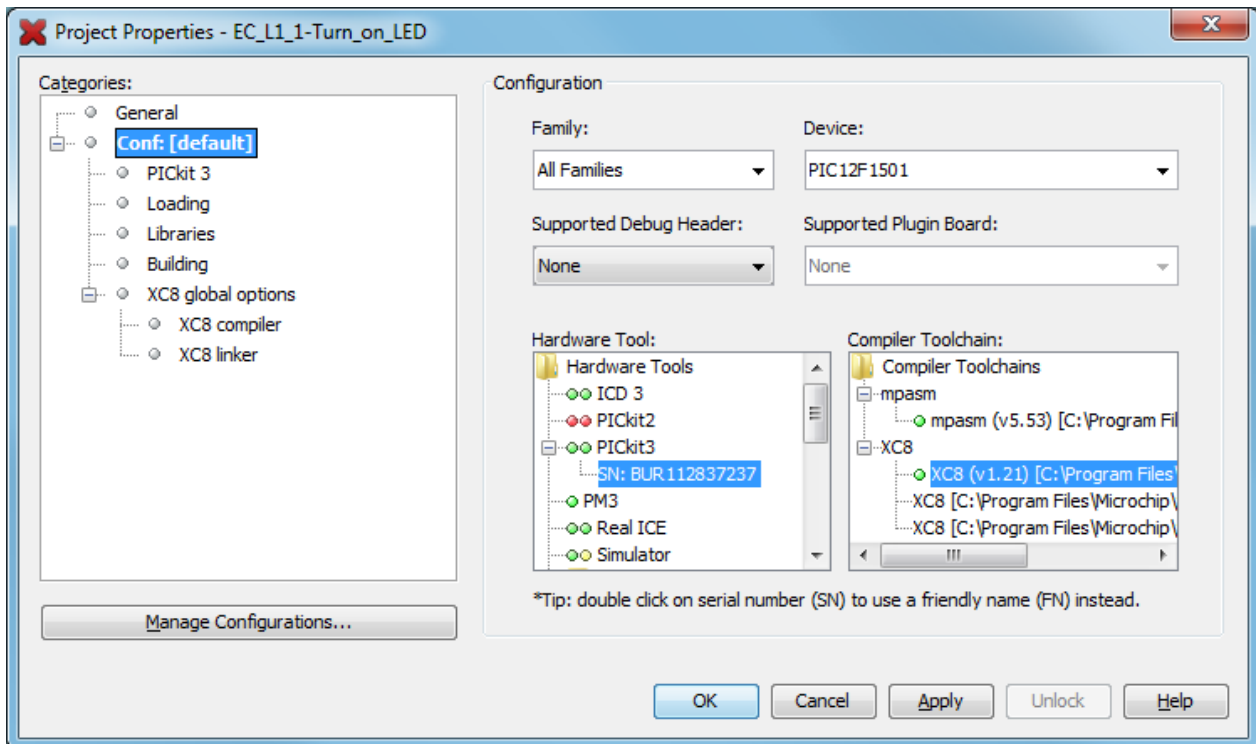
make[2]: Leaving directory `C:
make[1]: Leaving directory `C:

BUILD SUCCESSFUL (total time:
Loading code from C:/Work/Gool
Loading symbols from C:/Work/G
Loading completed

```

¹¹ Or, in general, that the PIC you wish to program is connected to whichever programmer or debugger you are using, whether it’s in a demo/development/training board, a production board, or a standalone programmer.

This will open the project properties window, where you can verify or change your hardware tool (programmer) selection:



After closing the project properties window, you can now program the PIC.

You can do this by right-clicking your project in the Projects window, and select “Make and Program Device”. This will repeat the project build, which we did earlier, but because nothing has changed (we have not edited the code), the “make” command will decide that there is nothing to do, and the compiler will not run.

Instead, in the “Build, Load” tab in the Output pane you should see output like:

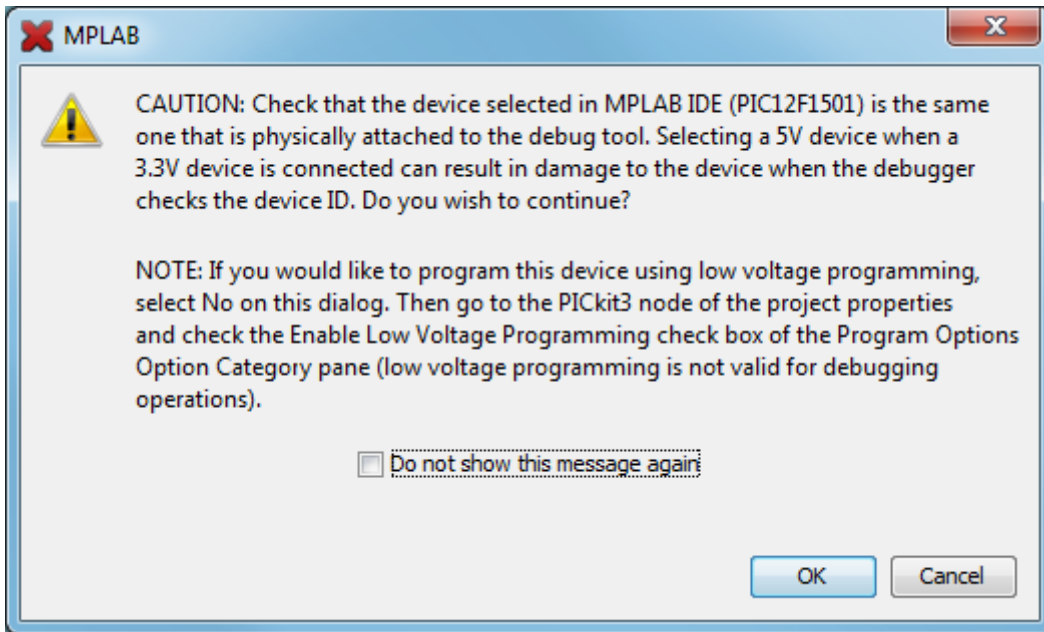
```
BUILD SUCCESSFUL (total time: 10s)
Loading code from C:/Work/Gooligum/Tutorials/Series 2/Web/Enhanced C/1 - Basic digital output/EC_L1_1-Turn_on_LED.X/dist/default/production/EC_L1_1-Turn_on_LED.X.production.hex...
Loading symbols from C:/Work/Gooligum/Tutorials/Series 2/Web/Enhanced C/1 - Basic digital output/EC_L1_1-Turn_on_LED.X/dist/default/production/EC_L1_1-Turn_on_LED.X.production.elf...Loading completed
Connecting to programmer...
Programming target...
```

A “PICKit 3” tab will also appear in the Output pane, where you can see what the PICKit 3 is doing.

Your PICKit 3 may need to have new firmware downloaded into it, to allow it to program enhanced mid-range devices, in which case you will see messages like:

```
Downloading Firmware...
Downloading RS...
Downloading AP...
AP download complete
Programming download...
Firmware Suite Version.....01.29.33
Firmware type.....Enhanced Midrange
```


You may also see a voltage caution warning, as shown below:



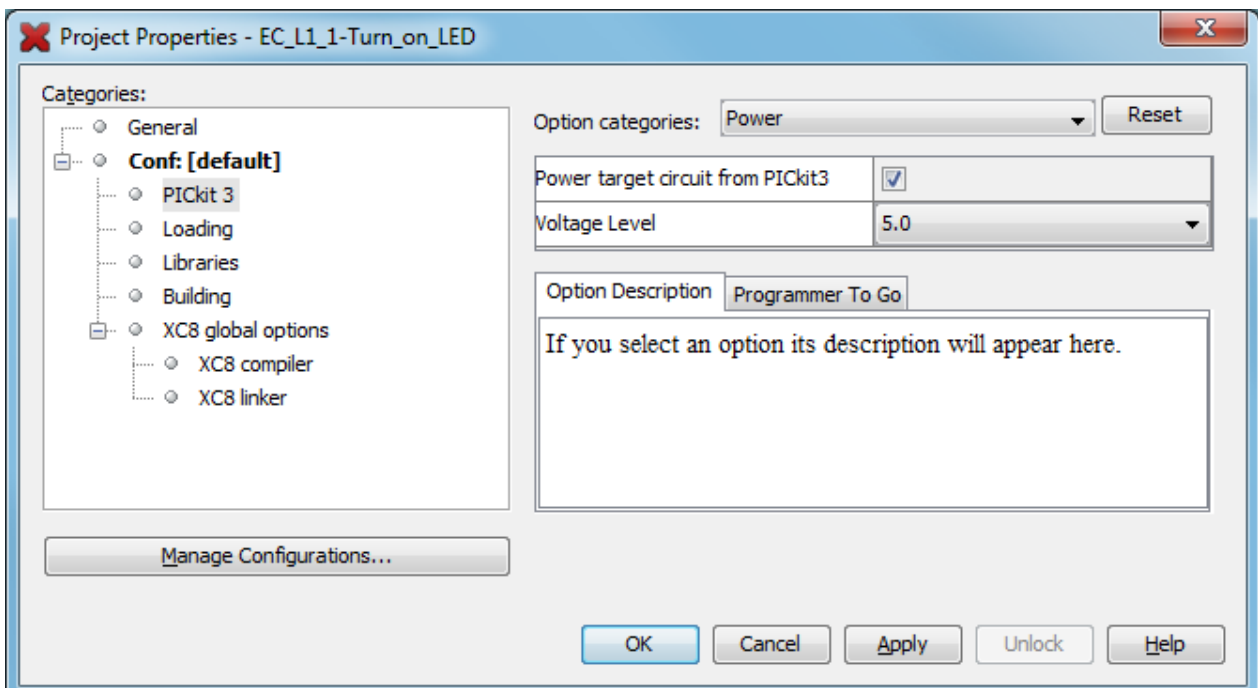
Since we are using a 5 V device, you can click 'OK'. And feel free to click "Do not show this message again", to avoid seeing this caution every time you program your PIC.

You may now see an error message in the PICKit 3 output tab, stating:

Target device was not found. You must connect to a target device to use PICKit 3.

This happens if the PIC is unpowered, so we need to tell the PICKit 3 to supply power.

Open the project properties window (as on the previous page), select 'PICKit 3' in the categories tree, and choose 'Power' option in the drop-down option categories list:



Select "Power target circuit from PICKit3", as shown. You can leave the voltage set to 5.0 V, and then click 'OK'.

If you now perform “Make and Program Device” again, the programming should be successful and you should see, in the build output tab, messages ending in:

Programming completed


Note that this action combines making (or building) the project, with programming the PIC.

In fact, there is no straightforward way with MPLAB X to simply program the PIC without building your project as well.

This makes sense, because you will almost always want to program your PIC with the latest code. If you make a change in the editor, you want to program that change into the PIC. With MPLAB X, you can be sure that whatever code you see in your editor window is what will be programmed into the PIC.

But most times, you’ll want to go a step further, and run your program, after uploading it into the PIC, to see if it works. For that reason, MPLAB X makes it very easy to build your code, program it into your PIC, and then run it, all in a single operation.

There are a few ways to do this:


- Right-click your project in the Projects window, and select “Run”, or
- Select the “Run → Run Main Project” menu item, or
- Press ‘F6’, or
- Click on the “Make and Program Device” button in the toolbar: 


Whichever of these you choose, you should see output messages ending in:

Running target...

The LED on RA1 should now light.

Being able to build, program and run in a single step, by simply pressing ‘F6’ or clicking on the “Make and Program Device” button is very useful, but what if you don’t want to automatically run your code, immediately after programming?

If you want to avoid running your code, click on the “Hold in Reset” toolbar button () before programming. You can now program your PIC as above.


Your code won’t run until you click the reset toolbar button again, which now looks like  and is now tagged as “Release from Reset”.

Summary

The sections above, on building your project and programming the PIC, have made using MPLAB X seem much more complicated than it really is.

Certainly, there are a lot of options and ways of doing things, but in practice it’s very simple.

Most of the time, you will be working with a single project, and only one hardware tool, such as a programmer or debugger, which you will have selected when you first ran the New Project wizard.

In that case (and most times, it will be), just press ‘F6’ or click on  to build, program and run your code – all in a single, easy step.

That’s all there is to it. Use the New Project wizard to create your project, add a template file to base your code on, use the editor to edit your code, and then press ‘F6’.

Conclusion

For such a simple task as lighting an LED, this has been a very long lesson!

In summary, we:

- Introduced the enhanced mid-range PIC architecture, using the PIC12F1501
- Looked at some PIC device configuration options
- Showed how to configure and use the PIC's output pins
- Implemented an example circuit using two development boards:
 - Gooligum training and development board
 - Microchip Low Pin Count Demo Board
- Introduced the XC8 compiler
- Showed how to use MPLAB X to:
 - Create a new C project, based on a template
 - Modify that template code
 - Build the program
 - Program the PIC, using a PICKit 3
 - Run the program

That does seem to be a lot of theory, to accomplish so little.

Nevertheless, after all this, you have a solid base to build on. You have a working development environment. You can create projects, modify your code, load (program) your code into your PIC, and make it run.

Congratulations! You've taken your first step in PIC development!

That first step is the hardest. From this point, we build on what's come before.

In the [next lesson](#), we'll make the LED flash...

Introduction to PIC Programming

Programming Enhanced Mid-Range PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 2: Flash an LED

In [lesson 1](#) we introduced Microchip's XC8 compiler and used it to write a program which lit a single LED connected to one of the pins of a PIC12F1501.

Now we'll make the LED flash.

In doing this, we will learn about:

- Selecting the internal oscillator frequency
- Using the XC8 delay function and macros

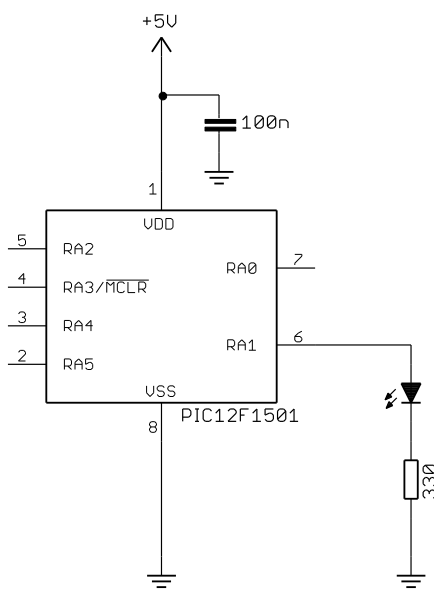
with examples implemented using XC8¹ (running in "Free mode").

The development environments and microcontrollers used for this lesson are the same as those in lesson 1.

Again, it is assumed that you are using a Microchip PICkit 3 programmer and either the [Gooligum Baseline and Mid-range PIC Training and Development Board](#) or Microchip's Low Pin Count (LPC) Demo Board, with XC 8 and Microchip's MPLAB X integrated development environment. But it is of course possible to adapt these instructions to a different programmers, compilers and/or development boards.

Example Circuit

Here's the circuit from [lesson 1](#) again:



If you have the Gooligum training board, simply plug the PIC12F1501 into the top section of the 14-pin IC socket – the section marked '12F'.

Connect a shunt across the jumper (JP12) on the LED labelled 'RA1', and ensure that every other jumper is disconnected.

If you are using Microchip's Low Pin Count Demo Board, refer back to [lesson 1](#) to see how to build this circuit, by soldering a resistor, LED (and optional isolating jumper) to the demo board, or by making connections on the demo board's 14-pin header.

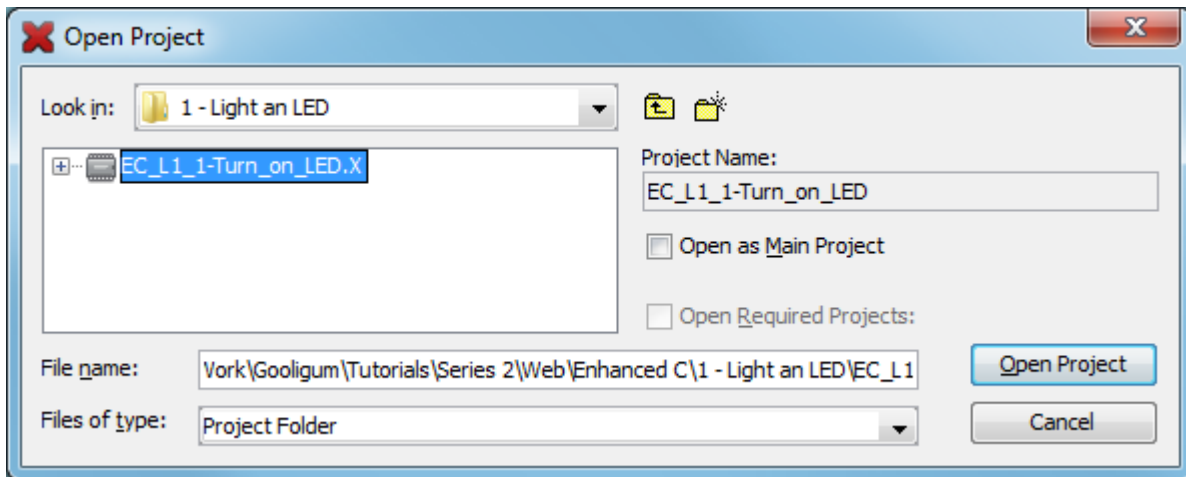
¹ Available as a free download from www.microchip.com.

Creating a new project in MPLAB X

It is a good idea, where practical, to base a new software project on work you've done before. In this case, it makes sense to build on the program from [lesson 1](#) – we just have to add extra statements to flash the LED.

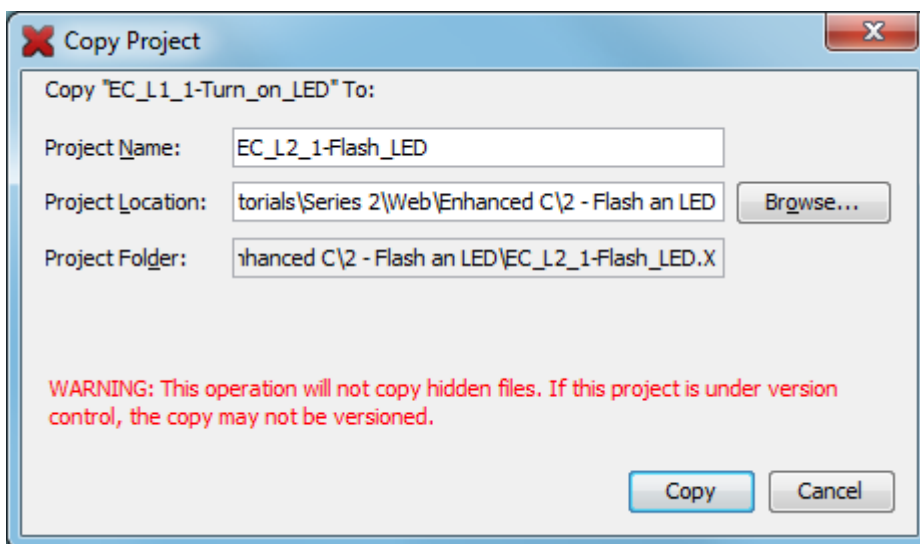
To create a new project in MPLAB X, based on an existing project, you first need to go into MPLAB X and open your existing project.

If you were recently working on the project you want to copy (such as the project from [lesson 1](#)), it is probably already visible in the Projects window. If it's not, it may appear under the “File → Open Recent Project” menu list. Or you can use the “File → Open Project” menu item, or click on the “Open Project...” toolbar button and browse to your project folder, select it, and click ‘Open Project’:



You should now right-click the project name ('EC_L1_1-Turn_on_LED' in this example) in the Projects window, and select “Copy...”.

The “Copy Project” dialog then gives you a chance to give your copied project a new name, such as 'EA_L2_1-Flash_LED'. You can also specify (and create, if you wish) a new folder for the project location, by browsing to it:

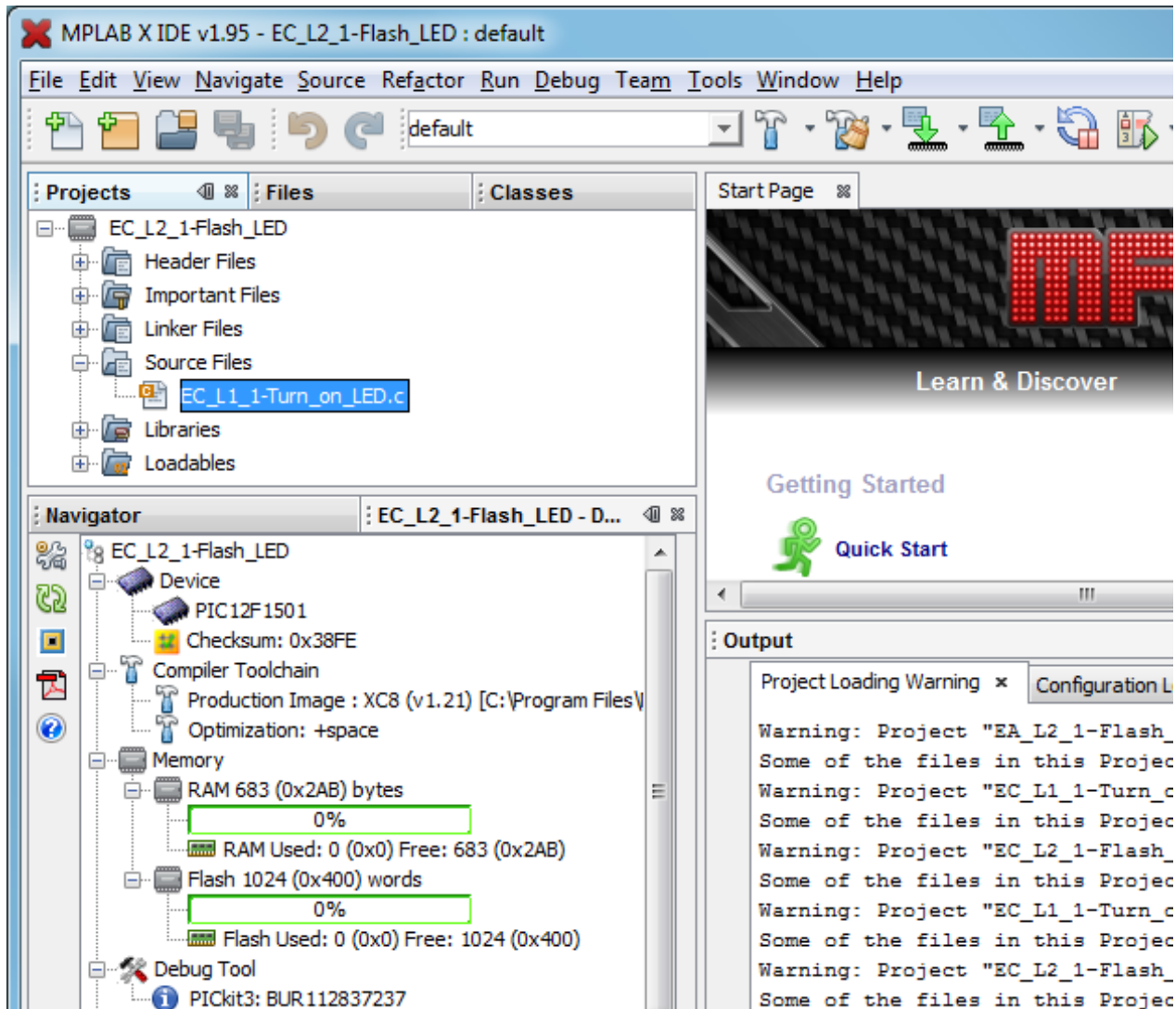


When you are satisfied with your new project name and location, click ‘Copy’.

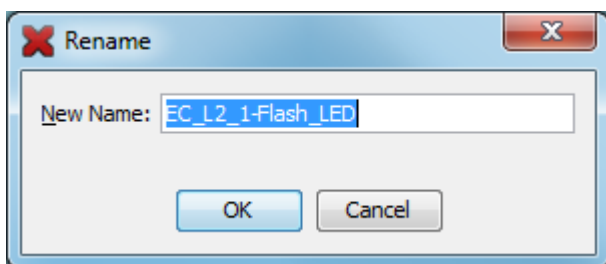
Your new project should now appear in the Projects window.

You can close your old project by right-clicking it and selecting “Close”, so that only your new project is visible.

If you expand your new project, you'll see that source file from the old project has been copied into the new project, with its original name:



To rename the source file, to something more appropriate for this project, right-click it and select "Rename...":

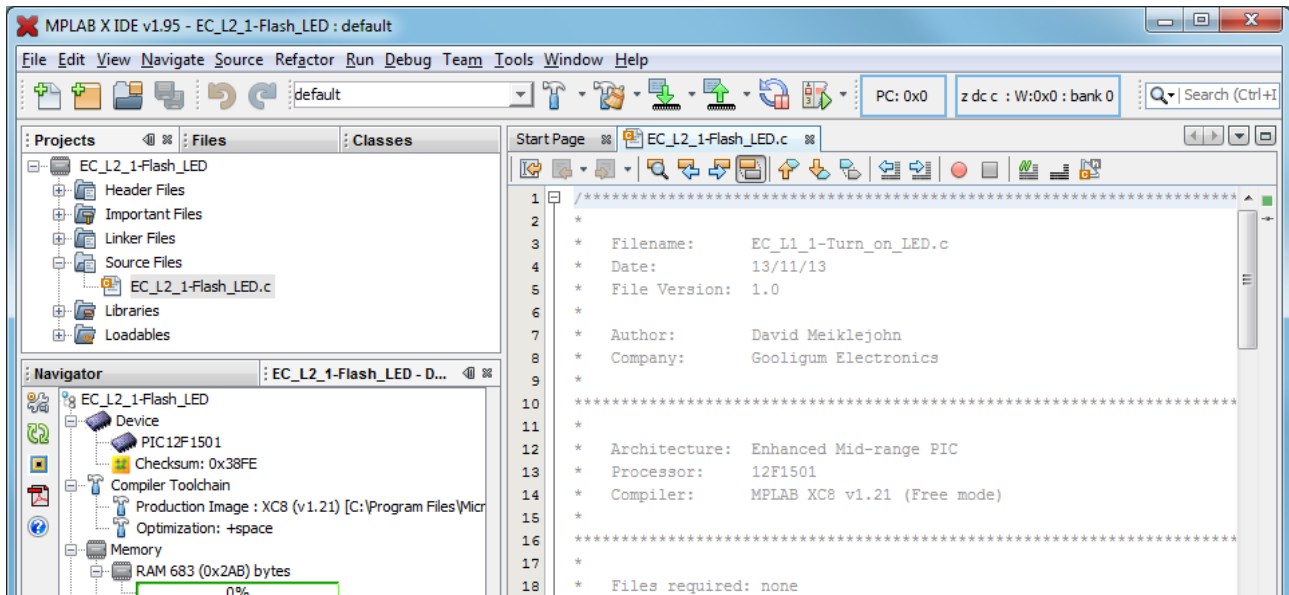


Type in the new name, such as 'EC_L2_1-Flash_LED' and then click 'OK'.

Note that there is no need to type the '.ASM' suffix – the Rename dialog will keep the existing file extension.

You now have a new project, with a new name in a new location, with a renamed source file, copied from your old project.

If you double-click your new source file, you'll see a copy of your code from [lesson 1](#) in an editor window:



Flashing the LED

You can now use the editor to update your code from [lesson 1](#).

We'll need to add some code to make the LED flash, but first the comments should be updated to reflect the new project. For example:

```

/*****
 *
 *  Filename:      EC_L2_1-Flash_LED.c
 *  Date:         20/11/13
 *  File Version: 1.0
 *
 *  Author:       David Meiklejohn
 *  Company:      Gooligum Electronics
 *
 *****/
 *
 *
 *  Architecture: Enhanced Mid-range PIC
 *  Processor:    12F1501
 *  Compiler:     MPLAB XC8 v1.21 (Free mode)
 *
 *****/
 *
 *  Files required: none
 *
 *****/
 *
 *  Description:   Lesson 2, example 1
 *
 *  Flashes an LED at approx 1 Hz.
 *  LED continues to flash until power is removed.
 *
 *****/
 *
 *  Pin assignments:
 *      RA1 = flashing LED
 *
 *****/

```

We're using the same PIC device as before, and it will be configured the same way, so we can leave the configuration pragmas unchanged.

And, as always in an XC8 program, we must begin by including the universal "xc.h" header file.

So we still have, unchanged from [lesson 1](#):

```
#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, internal oscillator (no clock out), no watchdog timer
#pragma config MCLRE = ON, FOSC = INTOSC, CLKOUTEN = OFF, WDTE = OFF
// brownout resets enabled, low brownout voltage, no low-power brownout reset
#pragma config BOREN = ON, BORV = LO, LPBOR = OFF
// no power-up timer, no code protect, no write protection
#pragma config PWRTE = OFF, CP = OFF, WRT = OFF
// stack resets on, high-voltage programming
#pragma config STVREN = ON, LVP = OFF
```

Again, we need to set up the PIC so that only RA1 is configured as an output, so we can leave the initialisation code from [lesson 1](#) intact:

```
/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    TRISA = ~(1<<1);           // configure RA1 (only) as an output
```

In [lesson 1](#), we made RA1 high, and left it that way. To make it flash, we need to set it high, then low, and then repeat.

You may think that you could achieve this with something like:

```
for (;;)
{
    LATAbits.LATA1 = 1;       // make RA1 high
    LATAbits.LATA1 = 0;       // make RA1 low
}                             // repeat forever
```

If you try this code, you'll find that the LED appears to remain on continuously.

In fact, it's flashing too fast for the eye to see – enhanced mid-range PICs are nowhere near the fastest microcontrollers available, but they are certainly fast enough to flash an LED thousands of times a second.

To slow it down enough to make the flashing visible, we have to add a delay.

XC8 provides a built-in function, '`_delay(n)`', which creates a delay 'n' instruction clock cycles long. The maximum possible delay depends on which PIC you are using, but it is a little over 50,000,000 cycles.

As mentioned in [lesson 1](#), the PIC has been configured to use its internal RC oscillator, which by default provides a 500 kHz processor clock. An instruction clock cycle corresponds to four processor clock cycles. So, with the default 500 kHz processor clock, corresponding to a 125 kHz instruction clock, that's a maximum delay of a little over 400 seconds.

The compiler also provides two macros: ‘`__delay_us()`’ and ‘`__delay_ms()`’, which use the ‘`__delay(n)`’ function to create delays specified in μ s and ms respectively. To do so, they reference the symbol “`_XTAL_FREQ`”, which you must define as the processor oscillator frequency, in Hertz.

So if our PIC is running at 500 kHz, we have:

```
#define _XTAL_FREQ 500000 // oscillator frequency for __delay()
```

Then, to generate a 500 ms delay, we can write:

```
__delay_ms(500); // stay on for 500 ms
```

We’ve mentioned a couple of times that the default processor clock speed is 500 kHz.

However, the PIC12F1501’s internal RC oscillators can be configured, via the `OSCCON` register, to provide a range of processor clock frequencies:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OSCCON	–	IRCF3	IRCF2	IRCF1	IRCF0	–	SCS1	SCS0

The `IRCF` bits are used to select the internal oscillator frequency, as follows:

<code>IRCF<3:0></code>	Oscillator	Frequency
000x	LF	31 kHz (approx)
001x	HF	31.25 kHz
0100	HF	62.5 kHz
0101	HF	125 kHz
0110	HF	250 kHz
0111	HF	500 kHz (default)
1011	HF	1 MHz
1100	HF	2 MHz
1101	HF	4 MHz
1110	HF	8 MHz
1111	HF	16 MHz

The 12F1501 actually has two internal RC oscillators: an uncalibrated low frequency oscillator, ‘`LFINTOSC`’, running at approximately 31 kHz, and a high frequency oscillator, ‘`HFINTOSC`’, which is factory-calibrated to run at 16 MHz.

This 16 MHz oscillator is used as the clock source in the remaining “HF” modes, divided by a postscaler to generate frequencies down as low as 31.25 kHz, as shown in the table on the left².

The internal clock source (`LFINTOSC` or `HFINTOSC`, as above) is selected whenever the `SCS1` bit is set, regardless of the processor configuration words.

Otherwise, if `SCS<1:0> = 00`, the clock source is selected by the oscillator selection bits in the configuration words.

The processor clock frequency isn’t really important in this example – any of these (even 31 kHz) is fast enough to flash an LED.

But it’s important to be aware of what frequency the processor is running at, so that you can correctly define the “`_XTAL_FREQ`” symbol. If you don’t get this right, your delays will be longer or shorter than expected!

² Not all possible `IRCF` values are shown here; those omitted duplicate some of the available processor frequencies.

And although we are using the default 500 kHz clock, it's good practice to explicitly initialise the oscillator in any program, such as this one, which assumes a specific processor frequency – your code will be more likely to work (or at least you'll see more easily what has to be changed) if you later move it to another processor.

So we should include in our initialisation routine:

```
// configure oscillator
OSCCONbits.SCS1 = 1;           // select internal clock
OSCCONbits.IRCF = 0b01111;    // internal oscillator = 500 kHz
```

Note again the use of bitfields (defined in the “pic12f1501.h” header file) to update individual bits such as **SCS1** or fields such as **IRCF** within a special function register such as **OSCCON**.

To make the LED flash at 1 Hz, with a duty cycle of 50% (500 ms on, then 500 ms off, and repeat), we could use as our main loop:

```
for (;;)
{
    LATAbits.LATA1 = 1;        // turn on LED on RA1

    __delay_ms(500);          // stay on for 500 ms

    LATAbits.LATA1 = 0;        // turn off LED on RA1

    __delay_ms(500);          // stay off for 500 ms
}                               // repeat forever
```

That will work, but it's possible to optimise this a little.

Instead of repeatedly setting and clearing **LATA1**, we could toggle it, with:

```
LATAbits.LATA1 = ~LATAbits.LATA1;
```

or:

```
LATAbits.LATA1 = !LATAbits.LATA1;
```

This works because single-bit bit-fields, such as **LATA1**, hold either a ‘0’ or ‘1’, representing ‘false’ or ‘true’ respectively, and so can be used with the logical negation operator ‘!’ – although the bitwise complement operator (~), used in the first version, is considered more appropriate when working with bits like this.

Note that in this example there is no need to set **RA1** to an initial state; whether it's high or low to start with, it will be successively flipped. But usually you will want to ensure that the output pins are in a known state before the main loop begins.

For example, if we wanted to begin with the LED off, we would clear the bit in **LATA** corresponding to **RA1**. We don't have any other output pins, but it doesn't hurt to clear the whole of **LATA**, in case the LED is moved to another pin or other LEDs added later.

In that case you would include in your initialisation code something like:

```
LATA = 0;                               // start with all output pins low (LED off)
```

It's usually best to initialise the output pins before they are configured as outputs, so that they do not, even for an instant, output an incorrect level when the program starts running.

So our port initialisation code becomes:

```
// configure port
LATA = 0;           // start with all output pins low (LED off)
TRISA = ~(1<<1);   // configure RA1 (only) as an output
```

Complete program

Putting together all these pieces, here's the complete LED flashing program:

```

/*****
 *
 * Description: Lesson 2, example 1
 *
 * Flashes an LED at approx 1 Hz.
 * LED continues to flash until power is removed.
 *
 *****/
 *
 * Pin assignments:
 * RA1 = flashing LED
 *
 *****/

#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, internal oscillator (no clock out), no watchdog timer
#pragma config MCLRE = ON, FOSC = INTOSC, CLKOUTEN = OFF, WDTE = OFF
// brownout resets enabled, low brownout voltage, no low-power brownout reset
#pragma config BOREN = ON, BORV = LO, LPBOR = OFF
// no power-up timer, no code protect, no write protection
#pragma config PWRTE = OFF, CP = OFF, WRT = OFF
// stack resets on, high-voltage programming
#pragma config STVREN = ON, LVP = OFF

#define _XTAL_FREQ 500000 // oscillator frequency for _delay()

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure port
    LATA = 0;           // start with all output pins low (LED off)
    TRISA = ~(1<<1);   // configure RA1 (only) as an output

    // configure oscillator
    OSCCONbits.SCS1 = 1; // select internal clock
    OSCCONbits.IRCF = 0b0111; // internal oscillator = 500 kHz

    /*** Main loop
    for (;;)
    {

```

```
// toggle LED on RA1
LATABits.LATA1 = ~LATABits.LATA1;

// delay 500 ms
__delay_ms(500);
} // repeat forever
}
```

If you follow the programming procedure described in [lesson 1](#), you should now see your LED flashing at something very close to 1 Hz.

Conclusion

It's taken two lessons and dozens of pages to get here, but we finally have a flashing LED!

In this lesson, we built on the first, showing how to base a new project on an existing one, modifying it and adding whatever additional features the new project needs.

We saw how to toggle a pin and select the processor clock speed.

We also saw how to use the delay macros provided by XC8.

In addition to providing an output (such as a blinking LED), PIC applications usually have to respond sensors and/or user input.

In the next lesson we'll see how to read and respond to switches, such as pushbuttons.

And since real switches “bounce”, which can be a problem for microcontroller applications, we'll look at ways to “debounce” them, in software.

Introductory 8-bit PIC Example Projects

Using C and Assembly Language

by David Meiklejohn, Gooligum Electronics

Project 1: Traffic Lights

This series of example projects for 8-bit PICs builds on the [Gooligum baseline](#) and [mid-range PIC assembly language](#) and [C tutorials](#), showing how real devices are developed, to further illustrate concepts introduced in the tutorials. As such, these example projects assume some familiarity with the material covered in the baseline and mid-range PIC tutorials, which will be referenced when appropriate.

The hardware for each project can be ordered in kit form (full or PCB-only) from the [Gooligum kit pages](#).

To get the most out of these examples, you should consider purchasing the [Gooligum Baseline and Mid-range PIC Training and Development Board](#), which includes all the lessons on CD. Alternatively, the tutorials can be ordered separately.

We'll assume that you have access to (and know how to use) a PIC development environment, as described in tutorial [lesson 0](#).

Although assembly language is used in some of these example projects, including this one, every project is also implemented in C, using Microchip's XC8 compiler¹ (running in "Free mode"). Some of the projects are only implemented in C, reflecting the fact that C is more widely used in embedded devices than assembly language – even in projects as simple as these ones.

Toy Traffic Lights

Traffic lights are fairly simple devices: a green light is on for some time, followed by an amber (yellow) light for a short time, and then a red light for what always feels like an eternity – and then the sequence continually repeats.

Of course real traffic lights are more complicated. Their controllers have some "smarts": the timing of the green/amber/red cycle depends on the time of day, and perhaps on whether a sensor has detected cars or a pedestrian has pressed the "cross" request button. They are often synchronised with other lights and may be centrally controlled by a traffic management authority. And they can be set to flash amber.

Toy traffic lights don't need to be so complicated. But even for a single, standalone set of lights, as we'll build in this project, could do with a little control. Sometimes you might want the lights to cycle automatically, just like real traffic lights. But other times you might want to be able to control them manually, perhaps pressing a button to advance the sequence from green to amber to red and so on. And that means that we'd also need to be able to select between automatic and manual operation.

Our toy traffic lights should be battery powered. We don't want the batteries to run flat, so we have to be able to power the lights on and off. Children (and adults...) often forget to turn their toys off, so ideally the traffic lights would also be able to power down automatically, if they haven't been used for some time. And that means that we need a way to power the lights back on, after they had shut themselves down.

¹ Available as a free download from www.microchip.com.

We know that PICs can enter a power-saving sleep mode (see e.g. [baseline assembler lesson 7](#)) and that they can be set to wake from sleep when an input changes. We can use sleep mode to implement the “power down automatically” requirement and wake-up on change for “power the lights back on”. And if we’re doing that, there’s no need for a separate power switch: if we need to have a button for “power on”, then we may as well use the same button for “power off”.

We’ve now identified an initial set of requirements:

- 3 × light outputs: green, yellow (amber) and red
- 1 × automatic/manual “mode-select” switch
- 1 × “change” pushbutton switch, to advance the lights in manual mode
- battery-powered
- low-power standby mode, with automatic timeout
- 1 × on/off pushbutton switch

We’ll still need to fill in some details, such as how long each light is on for in automatic mode, and how long the “power-off” timeout is.

Step 1: Simple automation only

When working on a project, even one as simple as this, it’s often best to proceed step by step – don’t try to design the whole thing at once, start by getting the core functions working, and be prepared to revise the design as you go.

We’ll start very simply, with just a set of three lights (green, yellow and red) that light automatically in turn, with no “smarts”.

We only need three outputs, and (at this stage) no inputs.

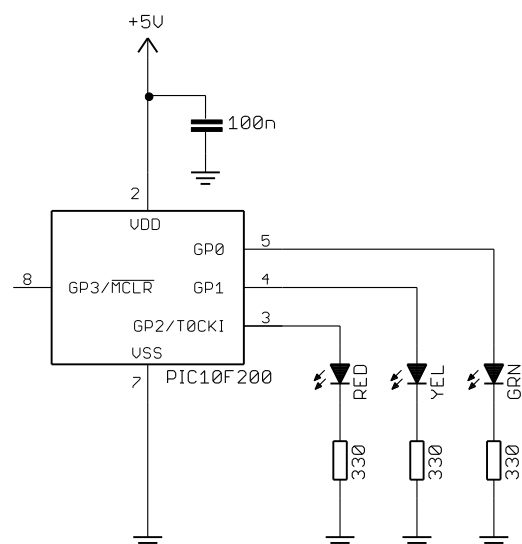
The smallest PIC that meets this requirement (indeed, the smallest PIC of all) is the 10F200, introduced in [baseline assembler lesson 1](#).

It has only three I/O pins, one input-only pin, 256 words of program memory, 16 bytes of data memory, no analog input capability, no advanced peripherals and only a single 8-bit timer (Timer0). But, for simply turning on three lights in sequence, even such a simple device is surely capable enough.

If we use ordinary LEDs as the lights, we can drive them directly from the PIC’s output pins, as shown in the diagram on the right.

Other than current-limiting resistors, a power supply and decoupling capacitor, that’s all we need.

Given standard intensity green, yellow and red LEDs, a 5 V power supply and 330 Ω resistors, the current through each LED will be around 10 mA, which is more than enough to light them brightly.



If you have the [Gooligum baseline training board](#), you can use it to implement this circuit.

Plug the PIC10F200 into the 8-pin IC socket marked '10F'.²

Connect shunts across jumpers JP11, JP12 and JP13 to connect the green LED to GP0, the yellow LED to GP1, and the red LED to GP2. Ensure that every other jumper is disconnected.

A PICkit 2 or PICkit 3 programmer can supply enough power for this circuit; there is no need to connect an external power supply.

The program is very simple – we can express it in pseudo-code as:

```
Initialisation:
    configure LED pins as outputs
    start with all LEDs off

Main loop:
do forever
    // light each LED in sequence
    turn on green
    delay for green "on" time
    turn off green

    turn on yellow
    delay for yellow "on" time
    turn off yellow

    turn on red
    delay for red "on" time
    turn off red
end
```

Whether you program in C or assembly language, your code will be more maintainable if you give the pins symbolic names, defined toward the start of your program (or in a header file), such as "G_LED" instead of "GP0". If you later change the connections – as we will as we develop this project – it is much easier to make the corresponding changes to your program code if you don't have to find and update every statement or instruction where that pin is referenced.

Similarly, you can make your code more maintainable by defining symbolic names for constants, such as "G_TIME" to represent the number of seconds that the green light should be turned on.

So, using symbolic definitions, our pseudo-code program becomes:

```
Definitions:
    G_LED = GP0          // LEDs
    Y_LED = GP1
    R_LED = GP2

    G_TIME = 12         // time (in seconds) each colour is turned on for
    Y_TIME = 3
    R_TIME = 10

Initialisation:
    configure LED pins as outputs
    start with all LEDs off
```

² Ensure that no device is installed in the 12F/16F socket – you can only use one PIC at a time in the training board.

```

Main loop:
do forever
    // light each LED in sequence
    G_LED = on           // green
    delay G_TIME secs
    G_LED = off

    Y_LED = on           // yellow
    delay Y_TIME secs
    Y_LED = off

    R_LED = on           // red
    delay R_TIME secs
    R_LED = off
end

```

XC8 implementation

This program is little more than flashing LEDs, which we saw how to do in C, using the XC8 compiler, in [baseline C lesson 1](#).

First, as we do for all XC8 programs, we include the 'xc.h' file which defines a number of macros and the symbols specific to our selected PIC device:

```
#include <xc.h>
```

We then configure the processor:

```

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog
#pragma config MCLRE = ON, CP = OFF, WDTE = OFF

```

Note that we've selected external reset, with the $\overline{\text{MCLR}}$ input enabled, even though no connection to the $\overline{\text{MCLR}}$ (GP3) pin is shown in the circuit diagram above. That's because the $\overline{\text{MCLR}}$ line is connected to your PIC programmer, allowing the programmer to reset the PIC. In a real design (which we'll get to...), you'd never leave any inputs floating – certainly not $\overline{\text{MCLR}}$ if external reset was enabled.

We'll be using the `__delay_ms()` delay macro, for which we need to define the oscillator frequency, which in this case is 4 MHz (the only possible frequency for a PIC10F200):

```

// oscillator frequency for __delay_ms()
#define _XTAL_FREQ 4000000

```

Completing the preliminaries, we can define the symbolic pin names and constants:

```

// Pin assignments
#define G_LED    GPIObits.GP0           // LEDs
#define Y_LED    GPIObits.GP1
#define R_LED    GPIObits.GP2

/***** CONSTANTS *****/
#define G_TIME    12                    // time (seconds) each colour is turned on for
#define Y_TIME    3
#define R_TIME    10

```


The main program, as always, begins with the `main()` function:

```

/***** MAIN PROGRAM *****/
void main()
{

```

We can then start program execution with an initialisation routine, to appropriately configure the PIC's I/O ports and peripherals:

```

    /*** Initialisation

    // configure ports
    GPIO = 0b0000;          // start with all LEDs off
    TRIS = 0b1000;         // configure LED pins (GP0-2) as outputs

    // configure timer
    OPTION = 0b11011111;   // configure Timer0:
                          //--0-----   timer mode (T0CS = 0)
                          //              -> GP2 usable as an output

```

Why configure the timer here? We're not actually going to use it, but as was explained in [baseline assembler lesson 5](#), the GP2 pin is not usable as an output by default, because at power-on it is configured as the Timer0 counter input. To make it possible to use GP2 as an output, we need to select timer mode. This is a common “gotcha” for beginners...

Most PIC programs consist of initialisation code (often encapsulated in separate functions), some interrupt service routines (not available in baseline PICs such as the 10F200, but see [mid-range assembler lesson 6](#) for an explanation) and an endlessly-repeating “main loop”, which may in turn call various functions.

So we next, and finally, have:

```

    /*** Main loop
    for (;;)
    {
        // light each LED in sequence
        G_LED = 1;          // turn on green LED
        __delay_ms(G_TIME*1000); // for green "on" time
        G_LED = 0;

        Y_LED = 1;          // turn on yellow LED
        __delay_ms(Y_TIME*1000); // for yellow "on" time
        Y_LED = 0;

        R_LED = 1;          // turn on red LED
        __delay_ms(R_TIME*1000); // for red "on" time
        R_LED = 0;

    }                          // repeat forever
}

```

Note that, because the `__delay_ms()` macro generates a delay in milliseconds, we need to multiply our delay times, such as `G_LED`, which we've specified in seconds, by 1000 to give the delay in milliseconds.

If we were going to have a lot of these, it might make sense to create a “`DelayS()`” macro which generates a delay in seconds, but it's not really worth doing that here.

MPASM implementation

To implement this program in assembly language, using the MPASM assembler, we'll draw on material from baseline assembler lessons [1](#), [2](#), [3](#), [5](#) and [6](#).

First, as in every MPASM program, we use the `list` directive to specify the processor type, and then include the appropriate header file to define processor-specific symbols:

```
list          p=10F200
#include      <p10F200.inc>
```

[Baseline assembler lesson 3](#) introduced the `banksel` and `pagesel` directives, used to overcome memory addressing limitations in the baseline PIC architecture in a portable, maintainable way. They're not actually applicable to the PIC10F200, which doesn't have multiple memory banks or pages. It's a good habit to use these directives anyway, to make it easy to move your code to a bigger device later, but the assembler will complain that they are not needed. We can stop it issuing those warnings with:

```
errorlevel -312 ; no "page or bank selection not needed" messages
```

We would be good to use the "DelayMS" macro developed in [baseline assembler lesson 6](#), which calls the "delay10" subroutine developed in [baseline assembler lesson 3](#). Unfortunately, unlike the XC8 equivalent, that macro can only generate delays up to 2.5 seconds – we'd need to call it multiple times.

An alternative is to create a new "delay1s" subroutine, to give a delay in seconds, based on "delay10", but with an extra loop:

```
*****
;
; Description:   Variable Delay : N x 1 seconds (1 - 255 secs)
;
; N passed as parameter in W reg
; exact delay = W x 1.0015 sec
;
; Returns: W = 0
; Assumes: 4 MHz clock
;
*****

#include      <p10F200.inc> ; any baseline device will do

errorlevel -312 ; no "page or bank selection not needed" messages

GLOBAL      delay1s_R

***** VARIABLE DEFINITIONS
          UDATA
dc1       res 1 ; delay loop counters
dc2       res 1
dc3       res 1
dc4       res 1

***** SUBROUTINES *****
          CODE

***** Variable delay: 1 to 255 seconds
;
```

```

; Delay = W x 1 sec
;
delay1s_R
    banksel dc4                ; delay = ?+1+Wx(2+1001499+3)-1+4 = W x 1.0015
sec
    movwf    dc4
dly3    movlw    .100           ; repeat middle loop 100 times
        movwf    dc3           ; -> 100x(3+10009+3)-1 = 1001499 cycles
dly2    movlw    .13           ; repeat inner loop 13 times
        movwf    dc2           ; -> 13x(767+3)-1 = 10009 cycles
        clrf     dc1           ; inner loop = 256x3-1 = 767 cycles
dly1    decfsz  dc1,f
        goto    dly1
        decfsz  dc2,f         ; end middle loop
        goto    dly1
        decfsz  dc3,f         ; end outer loop
        goto    dly2
        decfsz  dc4,f         ; end 1 sec count loop
        goto    dly3

    retlw    0

END

```

This code is then placed in a separate file, such as “delay1s.asm”, so that the subroutine can be called from our main program as an external module. To make this possible, the GLOBAL directive has been used to make the subroutine’s label, “delay1s_R”, externally accessible.

We can then encapsulate this subroutine within a macro, to make it easier to use:

```

;***** DelayS
; Delay in seconds
;
; Calls: 'delay1s' subroutine, providing a W x 1 sec delay
;
DelayS MACRO    secs                ; delay time in secs
    IF secs>.255
        ERROR "Maximum delay time is 255 secs"
    ENDIF
    movlw    secs
    pagesel delay1s
    call    delay1s
    pagesel $
    ENDM

```

If this macro is placed within an include file, such as “stdmacros-base.inc”, it can be made available to your program by “including” it toward the start of your main source file.

So, getting back to our main program, since we want to be able use our new “DelayS” macro, we add:

```

#include    <stdmacros-base.inc>    ; DelayS - delay in seconds
                                           ; (calls delay1s)
EXTERN    delay1s_R                ; W x 1 sec delay

```

The EXTERN directive is necessary, to allow our “DelayS” macro to call the “delay1s_R” subroutine, which is sitting in an external module.

Since, by default, the assembler interprets numeric constants as hexadecimal, which is a little counter-intuitive, you can make your life easier by changing the default radix to decimal, which is done with:

```
radix      dec
```

We can then configure the processor:

```
;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog
__CONFIG      _MCLRE_ON & _CP_OFF & _WDTE_OFF
```

As in the C example above, note that we've selected external reset, with the $\overline{\text{MCLR}}$ input enabled, even though no connection to the $\overline{\text{MCLR}}$ (GP3) pin is shown in the circuit diagram. What's not shown is that, on a development board, the $\overline{\text{MCLR}}$ line is connected to your PIC programmer, allowing the programmer to reset the PIC. It wouldn't be left floating like this in a real, final design.

Next we can define the symbolic pin names and constants:

```
; pin assignments
#define G_LED      GPIO,0      ; LEDs
#define Y_LED      GPIO,1
#define R_LED      GPIO,2

;***** CONSTANTS
constant G_TIME = 12          ; time (seconds) each colour is on for
constant Y_TIME = 3
constant R_TIME = 10
```

Before the main program commences, we update the internal RC oscillator calibration value to the factory setting, as usual:

```
;***** RC CALIBRATION
RCCAL    CODE    0x0FF          ; processor reset vector
        res 1                ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET    CODE    0x000          ; effective reset vector
        movwf   OSCCAL          ; apply internal RC factory calibration
        pagesel start
        goto    start          ; jump to main code
```

And then, to get around the baseline architecture's subroutine addressing limitation (see [baseline assembler lesson 3](#)), we have a subroutine jump table:

```
;***** Subroutine vectors
delay1s          ; delay W x 1 sec
        pagesel delay1s_R
        goto    delay1s_R
```

Strictly speaking, this precaution (using a jump table to call subroutines) is not necessary on the PIC10F200, which only has 256 words of program memory. But it may become necessary if we later move this code to a larger device, so we may as well include this now, to make any future migration easier.

The main part of the program starts with the initialisation routine, which configures the PIC's I/O ports and peripherals:

```

;***** MAIN PROGRAM *****
MAIN      CODE

;***** Initialisation
start
    ; configure port
    clrf   GPIO                ; start with all LEDs off
    movlw  b'1000'            ; configure LED pins (GP0-2) as outputs
    tris   GPIO
    ; configure timer
    movlw  b'11011111'        ; configure Timer0:
                                timer mode (T0CS = 0)
                                ; --0-----
    option                                ; -> GP2 usable as an output

```

Again, as we did in the C example, Timer0 is configured to use timer mode, making it possible to use the GP2 pin as an output, as explained in [baseline assembler lesson 5](#).

With the PIC configured, we come finally to the main loop:

```

;***** Main loop
main_loop
    ; light each LED in sequence
    bsf   G_LED                ; turn on green LED
    DelayS G_TIME              ; for green "on" time
    bcf   G_LED

    bsf   Y_LED                ; turn on yellow LED
    DelayS Y_TIME              ; for yellow "on" time
    bcf   Y_LED

    bsf   R_LED                ; turn on red LED
    DelayS R_TIME              ; for red "on" time
    bcf   R_LED

    ; repeat forever
    goto  main_loop

END

```

Our “DelayS” macro (and the “delay1s” subroutine which it calls) makes this main loop as short and simple as the C version was – turn on each LED, delay a certain number of seconds, turn off the LED, then do the same for each LED in sequence and continually repeat.

Step 2: Simple automation with sleep mode

The previous design was as simple as possible – just sequence the three lights. Now that that's working, we can start adding more features.

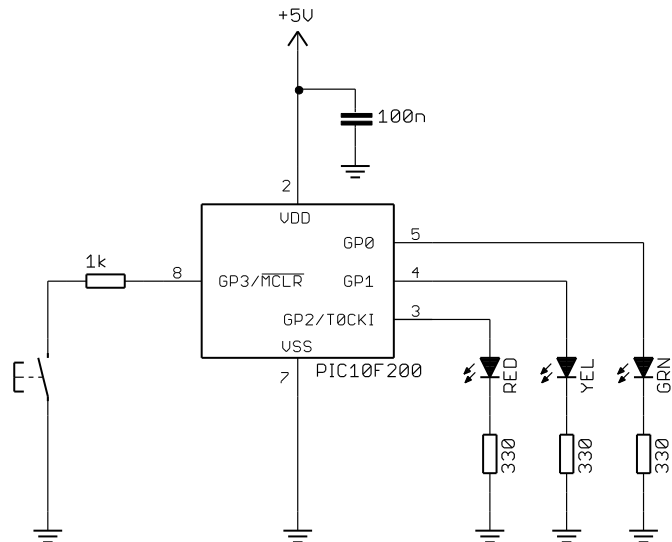
The first is the ability to turn the traffic lights on and off, by pressing a pushbutton.

As mentioned earlier, the “off” state won't really be fully off – it will be a “standby” state, using the PIC's low-power sleep mode, in which the PIC typically draws less than 1 μA . If the LEDs are turned off and there is negligible leakage in the rest of the circuit, the overall current consumption will also be less than 1 μA – low enough for our traffic lights to remain in standby mode for their batteries' entire shelf life.

We can connect the pushbutton to the GP3 pin, as shown on the right.

The 1 k Ω resistor isn't absolutely required, but as explained in [baseline assembler lesson 4](#), it's good practice to include an isolation resistor like this on inputs, especially on the GP3 pin when an in-circuit serial programming (ICSP) programmer, such as a PICkit 2 or PICkit 3, could be connected – this pin is also used for $\overline{\text{MCLR}}$ and the high programming voltage, and the isolation resistor helps to protect both the programmer and the PIC.

Note that there is no external pull-up resistor. Instead, we'll use the PIC's internal "weak pull-up" facility, as described in [baseline assembler lesson 4](#).



If you are using the [Gooligum baseline training board](#), you can leave it set up as for the previous circuit; it has a pushbutton switch already connected to GP3. There is no need to close any additional jumpers.

We'll need to disable external resets, to make it possible to use GP3 as an input.

The baseline PIC architecture does not support interrupts (see [mid-range assembler lesson 6](#)), so to detect pushbutton presses we'll need to poll GP3 within the main loop.

The need to poll GP3 is a problem (this would be easier if we had interrupts...).

Suppose we poll the button just once within of the main loop, for example (in pseudo-code):

```
do forever
    // light each LED in sequence
    turn on green
    delay for green "on" time
    turn off green

    turn on yellow
    delay for yellow "on" time
    turn off yellow

    turn on red
    delay for red "on" time
    turn off red

    // check for button press
    if button is pressed
        enter standby mode
end
```

The user may have to press the button for a long time (up to 25 seconds, assuming the delay values specified earlier) before the traffic lights detect the button press and respond by entering standby mode.

Even if we insert this "check for button press" code after every delay, the user may still have to press the button for 10 seconds or more (the green light delay is 12 seconds) before it is detected. That's terrible user interface design. We expect a device to respond to a button press in less than a second. In fact, we should aim for a response time of less than one tenth of a second – which would mean polling the pushbutton input at least ten times per second.

We can't do that if we have long, multiple-second delays, during which we don't detect or respond to inputs. So we'll need to rethink our approach. One option would be to switch to a similar low-end mid-range PIC, such as the 10F320, to be able to use interrupts. But there's no real need – we can do this easily enough using the baseline architecture.

To generate a long delay, we can use a sequence of short delays, polling the pushbutton between each.

Instead of repeating this “polling delay” code multiple times, we can restructure the main loop (using pseudo-code) as follows:

```
do forever
  // light each LED in sequence, while checking for button press
  for seconds = 0 to end_cycle_time
    // light appropriate LED, depending on elapsed time
    if seconds = start_green_time
      turn off all
      turn on green

    if seconds = start_yellow_time
      turn off all
      turn on yellow

    if seconds = start_red_time
      turn off all
      turn on red

    // delay 1 second while polling pushbutton
    repeat 1000/N times
      delay N ms
      // check for button press
      if button pressed
        enter standby mode
  end
end
```

This code uses a seconds counter to keep track of what happens when in the traffic light cycle. When the counter reaches various predetermined values, the appropriate LED is turned on (all other LEDs are turned off). At the end of the cycle (when the red light finishes) the loop is restarted.

Note that the “polling delay” loop has been specified in a way that is independent of the inner delay length, which is specified only as “N ms”. If the inner delay is only 10 ms, $N = 10$ and the outer loop executes $1000/10 = 100$ times.

The timing of this “1 second” polling loop won't be exact, because it doesn't take the polling overhead into account. As the inner delay becomes shorter, the polling overhead becomes comparatively greater. As the inner delay is made longer, the polling interval increases, making the pushbutton less responsive. A reasonable compromise is a 50 ms delay.

Adding symbolic definitions, our pseudo-code program becomes:

```
Definitions:
  LEDS    = GPIO    // all LEDs
  G_LED   = GP0     // individual LEDs
  Y_LED   = GP1
  R_LED   = GP2
  BUTTON  = GP3     // pushbutton

  G_TIME  = 12      // time (in seconds) each colour is turned on for
  Y_TIME  = 3
  R_TIME  = 10
```

```

G_START = 0          // seconds into cycle to turn on each LED
Y_START = G_TIME
R_START = Y_START + Y_TIME

R_END    = R_START + R_TIME    // total cycle length

POLL_MS = 50            // polling interval (in ms)

Initialisation:
// configure hardware
configure LED pins as outputs
start with all LEDs off
enable internal pull-ups
enable wake-up on change

// ensure that pushbutton is not pressed
wait for BUTTON = released
debounce BUTTON

Main loop:
do forever
// light each LED in sequence
for sec_cnt = 0 to R_END-1
// light appropriate LED, depending on elapsed time
if sec_cnt = G_START
LEDS = off
G_LED = on          // green

if sec_cnt = Y_START
LEDS = off
G_YED = on         // yellow

if sec_cnt = R_START
LEDS = off
R_LED = on         // red

// delay 1 second while polling pushbutton
repeat 1000/POLL_MS times
delay POLL_MS ms
// check for button press
if BUTTON = pressed
debounce BUTTON
enter standby mode
end
end

```

Note that we've added a section to the initialisation routine to ensure that the pushbutton is not pressed when the main loop begins. As explained in [baseline assembler lesson 7](#), this is necessary in case the pushbutton had been pressed to wake the device from sleep; if it's still pressed when we get to the test at the end of the main loop, it will be seen as a "new" button press and the device will go into standby mode. Similarly, it is important to debounce the pushbutton press before entering standby mode, to ensure that switch bounce doesn't count as a "change" and wake the device from sleep.

Note also that the seconds count finishes at "R_END-1", i.e. one less than the total cycle time (when the red light finishes) in seconds. That's because the count starts at zero, not one, so the total number of iterations through the `for` loop will be equal to the cycle time.

XC8 implementation

To implement this step's additional features in C, we'll draw on the explanations of reading switches and using the internal pull-ups in [baseline C lesson 2](#), the timer-based switch debounce method from [baseline C lesson 3](#), and the material on sleep mode and wake-up on change from [baseline C lesson 4](#).

First, we include not only 'xc.h' as usual, but also 'stdint.h' to define the standard 'uint8_t' type (see [baseline C lesson 1](#)) that we'll be using for the variables, as well as the 'stdmacros-XC8.h' file, which defines various useful macros that we've developed for XC8 :

```
#include <xc.h>
#include <stdint.h>

#include "stdmacros-XC8.h" // DbnceHi() - debounce switch, wait for high
                          // Requires: TMR0 at 256 us/tick
```

Note that 'xc.h' and 'stdint.h' are enclosed in '<>', because they are standard header files provided by the compiler and located in the compiler's 'include' directory, while 'stdmacros-XC8.h' is enclosed in '"', because it's a file that we've created, located locally, in our project directory.

The 'stdmacros-XC8.h' file contains the 'DbnceHi()' macro developed in [baseline C lesson 3](#):

```
#define DEBOUNCE 10*1000/256 // switch debounce count = 10 ms/(256us/tick)

// DbnceHi()
//
// Debounce switch on given input pin
// Waits for switch input to be high continuously for DEBOUNCE*256/1000 ms
//
// Uses: TMR0          Assumes: TMR0 running at 256 us/tick
//
#define DbnceHi(PIN) TMR0 = 0;          /* reset timer          */ \
                    while (TMR0 < DEBOUNCE) /* wait until debounce time */ \
                        if (PIN == 0)      /* if input low,         */ \
                            TMR0 = 0      /* restart wait          */ \
```

The processor is configured similarly to before, except that we need to disable the external reset function, to allow GP3 to be used as an input:

```
/* ***** CONFIGURATION ***** */
// int reset, no code protect, no watchdog
#pragma config MCLRE = OFF, CP = OFF, WDTE = OFF
```

In addition to the LED symbols we used in the first step, we'll define symbols to represent include the pushbutton:

```
#define BUTTON GPIObits.GP3 // Pushbutton (active low)
```

and also a symbol to represent all of the LEDs, so that we can turn them all off in a single operation³:

```
#define LEDS GPIO // all LEDs
```

³ This is only possible because all of the LEDs are on a single I/O port; we wouldn't be able to do it this way if the LEDs were connected to multiple ports on a larger PIC.

We need to add constants to represent the start time for each LED, as well as the overall cycle time:

```
#define G_START 0 // seconds into cycle to turn on each LED
#define Y_START G_TIME
#define R_START Y_START + Y_TIME

#define R_END R_START + R_TIME // total cycle length
```

We'll also define the polling interval as a constant:

```
#define POLL_MS 50 // polling interval (in ms)
```

At the start of the main() function, we declare the local variables that we will be using:

```
void main()
{
    uint8_t sec_cnt; // seconds counter
    uint8_t p_cnt; // polling loop counter
```

The initialisation code is similar to before, except that we also need to specify OPTION register bits to configure Timer0, weak pull-ups and wake-up on change:

```
// configure wake-on-change and timer
OPTION = 0b00000111; // configure wake-up on change and Timer0:
//0----- enable wake-up on change (/GPWU = 0)
//-0----- enable weak pull-ups (/GPPU = 0)
/--0----- timer mode (T0CS = 0)
/----0---- prescaler assigned to Timer0 (PSA = 0)
//-----111 prescale = 256 (PS = 111)
// -> increment every 256 us
// GP2 usable as an output
```

As discussed earlier, we ensure that the pushbutton is released (and no longer bouncing) before entering the main loop, in case the device has been woken from sleep by a pushbutton press:

```
// wait for stable button release
// (in case it is still bouncing following wake-up on change)
DbncHi(BUTTON);
```

The main loop is then a fairly straightforward translation into C of the pseudo-code version, above:

```
/***/ Main loop
for (;;)
{
    // light each LED in sequence
    for (sec_cnt = 0; sec_cnt < R_END; sec_cnt++)
    {
        // light appropriate LED, depending on elapsed time
        if (sec_cnt == G_START)
        {
            LEDES = 0; // turn off all LEDs
            G_LED = 1; // turn on green LED
        }
        if (sec_cnt == Y_START)
        {
            LEDES = 0; // turn off all LEDs
```

```

        Y_LED = 1;          // turn on yellow LED
    }
    if (sec_cnt == R_START)
    {
        LEDES = 0;          // turn off all LEDs
        R_LED = 1;          // turn on red LED
    }

    // delay 1 second while polling pushbutton
    // (repeat 1000/POLL_MS times)
    for (p_cnt = 0; p_cnt < 1000/POLL_MS; p_cnt++)
    {
        __delay_ms(POLL_MS);    // polling interval

        // check for button press
        if (!BUTTON)
        {
            // go into standby (low power) mode
            LEDES = 0;          // turn off all LEDs
            DbnceHi(BUTTON);    // wait for stable button release
            SLEEP();            // enter sleep mode
        }
    }
}
// repeat forever
}

```

Again, as mentioned, the pushbutton is debounced before entering sleep mode, to ensure that switch bounce doesn't immediately wake the device.

MPASM implementation

To implement this step in assembly language, we'll draw on the explanations of reading switches and using the internal pull-ups in [baseline assembler lesson 4](#), the timer-based switch debounce method from [baseline assembler lesson 5](#), and sleep mode and wake-up on change from [baseline assembler lesson 7](#).

As before, we'll include the 'stdmacros-base.inc' file which contains the definitions of the macros we wish to use, such as the 'DbnceHi' switch debounce macro developed in [baseline assembler lesson 6](#):

```

;***** DbnceHi
; Debounce switch on given input port,pin
; Waits for switch to be 'high' continuously for 10 ms
;
; Uses:      TMR0          Assumes: TMR0 running at 256 us/tick
;
DbnceHi MACRO    port,pin
    local        start,wait,DEBOUNCE
    variable     DEBOUNCE=.10*.1000/.256 ; switch debounce count =
                                           ; 10ms/(256us/tick)

    pagesel $          ; select current page for gotos
start    clrf          TMR0          ; button down so reset timer (counts "up" time)
wait     btfss         port,pin     ; wait for switch to go high (=1)
         goto          start
         movf          TMR0,w        ; has switch has been up continuously for
         xorlw         DEBOUNCE     ; debounce time?
         btfss        STATUS,Z      ; if not, keep checking that it is still up
         goto          wait
        ENDM

```

It also contains the definition of the 'DelayMS' macro developed in [baseline assembler lesson 6](#), which in turn calls the 'delay10' subroutine developed in [baseline assembler lesson 3](#). They are very similar to the 'DelayS' macro and 'delay1s' subroutine we used in the first step, so there is no need to list them here. Again, the delay code is placed in a separate "delay10.asm" file, linked with our main program, and made accessible via GLOBAL and EXTERN directives.

So, the start of our main program becomes:

```
list          p=10F200
#include      <p10F200.inc>

errorlevel   -312      ; no "page or bank selection " messages

#include      <stdmacros-base.inc> ; DbnceHi - debounce sw, wait for high
                                           ; (requires TMR0 running at 256 us/tick)
                                           ; DelayMS - delay in milliseconds
                                           ; (calls delay10)
EXTERN       delay10_R      ; W x 10ms delay

radix        dec
```

Since we need to disable external resets, allowing GP3 to be used as an input, the processor configuration becomes:

```
;***** CONFIGURATION
                ; int reset, no code protect, no watchdog
__CONFIG       _MCLRE_OFF & _CP_OFF & _WDTE_OFF
```

As we did in the C version, we'll define additional symbols to represent the pushbutton:

```
#define BUTTON      GPIO,3      ; Pushbutton (active low)
```

and also a symbol to represent all of the LEDs, so that we can turn them all off in a single operation:

```
#define LEDS        GPIO        ; all LEDs
```

And again we will add constants to represent the start time for each LED, as well as the overall cycle time:

```
constant G_START = 0                ; seconds into cycle to turn on each LED
constant Y_START = G_TIME
constant R_START = Y_START + Y_TIME

constant R_END = R_START + R_TIME    ; total cycle length
```

and define the polling interval as a constant:

```
constant POLL_MS = 50                ; polling interval (in ms)
```

We also need to define the variables that we will be using:

```
;***** VARIABLE DEFINITIONS
VARS  UDATA
sec_cnt  res 1                ; seconds counter
p_cnt    res 1                ; polling loop counter
```

The initialisation code is similar to that in the first step, except that we also need to specify OPTION register bits to configure Timer0, weak pull-ups and wake-up on change:

```

; configure wake-on-change, pull-ups and timer
movlw    b'00000111'    ; configure wake-up on change and Timer0:
; 0-----            enable wake-up on change (/GPWU = 0)
; -0-----            enable weak pull-ups (/GPPU = 0)
; --0-----           timer mode (T0CS = 0)
; ----0---            prescaler assigned to Timer0 (PSA = 0)
; -----111          prescale = 256 (PS = 111)
option   ;                -> increment every 256 us
;                GP2 usable as an output

```

And again, we have to ensure that the pushbutton is released (and no longer bouncing) before starting the main loop:

```

; wait for stable button release
; (in case it is still bouncing following wake-up on change)
DbncHi  BUTTON

```

The main loop is a translation of the ‘for’ loop from the pseudo-code version, where we initialise the seconds counter and then for each time through the loop we compare the counter against the various LED start times, lighting LEDs as appropriate, before delaying 1 second (while polling the pushbutton) then incrementing the seconds count and, if we’re not at the end of the cycle yet, repeating the loop.

So, at the start of the main loop, before our ‘for’ loop begins, we zero the seconds counter:

```

main_loop
; initialise seconds count (used to light each LED in sequence)
banksel sec_cnt    ; sec_cnt = 0
clrf    sec_cnt

```

Then within the “automatic light sequencing” loop, we compare the current count against the LED start times, and light one of the LEDs if the count matches:

```

auto_loop
;*** Light appropriate LED, depending on elapsed time
banksel sec_cnt
movf    sec_cnt,w    ; if sec_cnt = G_START
xorlw   G_START
btfss  STATUS,Z
goto   auto_yellow
clrf   LEADS        ; turn off all LEDs
bsf    G_LED        ; turn on green LED
auto_yellow
movf    sec_cnt,w    ; if sec_cnt = Y_START
xorlw   Y_START
btfss  STATUS,Z
goto   auto_red
clrf   LEADS        ; turn off all LEDs
bsf    Y_LED        ; turn on yellow LED
auto_red
movf    sec_cnt,w    ; if sec_cnt = R_START
xorlw   R_START
btfss  STATUS,Z
goto   auto_red_end
clrf   LEADS        ; turn off all LEDs
bsf    R_LED        ; turn on red LED
auto_red_end

```

Next comes the 1-second delay loop, during which we poll the pushbutton switch:

```

    ;*** Delay 1 second while polling pushbutton
    banksel p_cnt
    movlw 1000/POLL_MS    ; loop 1s/(POLL_MS/loop) times
    movwf p_cnt
poll_loop
    DelayMS POLL_MS      ; polling interval
    ; check for button press
    btfss BUTTON         ; if button down (low)
    goto standby         ; go into standby mode
    decfsz p_cnt,f
    goto poll_loop

```

When a pushbutton press is detected, the code jumps to a separate “enter standby mode” routine, placed at the end of the program (i.e. after the end of the main loop):

```

;***** Standby (low power) mode
standby
    clrf  LEDS          ; turn off LEDs
    DbnceHi BUTTON     ; wait for stable button release
    sleep              ; enter sleep mode

    END

```

Note that this “enter standby” routine could instead have been incorporated within the polling loop:

```

    ;*** Delay 1 second while polling pushbutton
    banksel p_cnt
    movlw 1000/POLL_MS    ; loop 1s/(POLL_MS/loop) times
    movwf p_cnt
poll_loop
    DelayMS POLL_MS      ; polling interval
    ; check for button press
    btfss BUTTON         ; if button down (low)
    goto btn_no_press    ; go into standby mode:
    clrf  LEDS          ; turn off LEDs
    DbnceHi BUTTON     ; wait for stable button release
    sleep              ; enter sleep mode
btn_no_press
    decfsz p_cnt,f
    goto poll_loop

```

Although this is closer in structure to the C version, the code version seems easier to follow if the “enter standby” routine is brought out as a separate routine, instead of being buried in the polling loop like this.

At the end of the light sequencing loop we increment the seconds count. The loop repeats until the cycle is finished (at the end of the red light), at which time we restart the main loop to start the cycle again:

```

;*** End seconds count loop
banksel sec_cnt
incf sec_cnt,w          ; sec_cnt = sec_cnt+1
movwf sec_cnt
xorlw R_END             ; loop until sec_cnt = R_END
btfss STATUS,Z
goto auto_loop

;*** Repeat forever
goto main_loop

```

Step 3: Adding a timeout

To save batteries, our traffic lights should automatically turn themselves off after a certain time.

It's quite simple to add this feature to the previous design: we need to add a time counter, which is incremented within the sequencing loop, keeping track of how long the lights have been operating. When the counter reaches the predetermined timeout value (say, 10 minutes), the device enters standby, in the same way as if the pushbutton had been pressed. Since wake-up on change is enabled, pressing the pushbutton will still wake the device from sleep, regardless of whether it had entered sleep through a timeout or button press.

To make our program more maintainable, we should define the timeout value as a symbolic constant.

A key decision in programming is often how to represent values such as “how long the lights have been operating” – should it be a single variable measuring seconds, or perhaps two variables storing minutes and seconds separately? Which representation you select will depend on factors such as the programming language you are using (some approaches make more sense in C than assembly language), or what else you might use the value for.

Nevertheless, when expressing the program in pseudo-code, we don't necessary need to make that decision up front, leaving the implementation details for later.

So we can define the timeout value simply as:

```
TIMEOUT = 10 // auto-off timeout (in minutes)
```

We'll need a time counter which is initialised (zeroed) when the program starts.

We then need to insert code to increment this counter and compare it against the timeout value within the light sequencing loop, as follows:

```
Initialisation:
    // configure hardware

    // ensure that pushbutton is not pressed

    // initialise time count
    time_cnt = 0

Main loop:
do forever
    // light each LED in sequence
    for sec_cnt = 0 to R_END-1
        // light appropriate LED, depending on elapsed time

        // delay 1 second while polling pushbutton

        // increment time count and check for timeout
        time_cnt = time_cnt+1
        if time_cnt = TIMEOUT*60 // timeout in seconds
            enter standby mode
    end
end
```

Note that the time counter is assumed here to be a single variable holding the number of seconds since the device was reset, so it is compared with the timeout value converted to seconds. But, as mentioned, we might choose to implement the time counter as separate minutes and seconds, in which case we'd only compare the minutes part with the timeout value.

XC8 implementation

We don't need any new PIC programming techniques to implement the timeout feature in C – we only have to add an extra constant definition, a variable, and a test.

Firstly, we add the timeout value to our constant definitions:

```
#define TIMEOUT 10 // auto-off timeout (in minutes)
```

We'll also need to declare the time counter as a variable. Since we'll be counting seconds, and a timeout of 10 minutes is 600 seconds, and 8-bit variables can only hold values up to 255, we'll need a 16-bit variable:

```
uint16_t time_cnt = 0; // timeout counter (seconds since reset)
```

An unsigned 16-bit integer can hold values up to 65535, so the maximum possible timeout period will be 65535 seconds = 1029 minutes and 15 seconds, or 18.2 hours. That should be plenty.

Note that the count is zeroed as part of its declaration; we don't need to do it separately in the initialisation code.

Finally, we add the timeout test at the end of the LED sequencing loop:

```
// check for timeout
if (++time_cnt == TIMEOUT*60)
{
    // go into standby (low power) mode
    LEDS = 0; // turn off all LEDs
    SLEEP(); // enter sleep mode
}
```

Note that the '+' operator is used to increment the time counter before it is compared with the timeout value (which is converted to seconds).

This saves a line of code, while still being clear to someone familiar with C.

We could leave it there, but consider that we now have “enter standby mode” code in two places – after the pushbutton test, and after the timeout test.

When similar chunks of code are repeated in different parts of a program, it may make sense to replace them with a function call, or perhaps a macro.

In this case, although the two pieces of code are not exactly the same, because there is no need to wait for a pushbutton release after detecting a timeout, it doesn't hurt to use the same “enter standby” code in both cases – there is no problem with waiting for a pushbutton release when a timeout is detected, because the pushbutton will already be released (if we've detected a timeout, the button cannot have been pressed).

So we can define an “enter standby” function:

```
***** FUNCTIONS *****/

***** Enter standby (low power) mode *****/
void standby(void)
{
    LEDS = 0; // turn off all LEDs
    DbncHi(BUTTON); // wait for stable button release
    SLEEP(); // enter sleep mode
}
```


And, since we usually place function definitions at the end of the program, we need to add a prototype for it, before `main()`:

```

/***** PROTOTYPES *****/
void standby(void);           // enter standby (low-power) mode

```

We can then call this function, to place the device in standby mode, after the pushbutton and timeout tests.

The LED sequencing loop then becomes:

```

// light each LED in sequence
for (sec_cnt = 0; sec_cnt < R_END; sec_cnt++)
{
    // light appropriate LED, depending on elapsed time
    if (sec_cnt == G_START)
    {
        LEDES = 0;           // turn off all LEDs
        G_LED = 1;           // turn on green LED
    }
    if (sec_cnt == Y_START)
    {
        LEDES = 0;           // turn off all LEDs
        Y_LED = 1;           // turn on yellow LED
    }
    if (sec_cnt == R_START)
    {
        LEDES = 0;           // turn off all LEDs
        R_LED = 1;           // turn on red LED
    }

    // delay 1 second while polling pushbutton
    // (repeat 1000/POLL_MS times)
    for (p_cnt = 0; p_cnt < 1000/POLL_MS; p_cnt++)
    {
        __delay_ms(POLL_MS); // polling interval

        // check for button press
        if (!BUTTON)
            standby();       // enter standby mode
    }

    // check for timeout
    if (++time_cnt == TIMEOUT*60)
        standby();         // enter standby mode
}

```

MPASM implementation

We can apply techniques we've already used to implement the timeout feature in assembly language.

We can start by adding the timeout value to our constant definitions:

```

constant TIMEOUT = 10           ; auto-off timeout (in minutes)

```

We also need to keep track of the time. As explained for the C version, if we wish to store a single value holding the number of seconds since the device was reset, it needs to be a 16-bit (2-byte) variable:

```

time_cnt    res 2           ; timeout counter (seconds since reset)

```

Unlike C, where we can initialise a variable as part of its declaration, when programming in assembly language we need to explicitly load initial values into variables as part of our initialisation code:

```
; initialise variables
banksel time_cnt
clrf    time_cnt      ; time_cnt = 0
clrf    time_cnt+1
```

This time counter is incremented after the 1-second delay in our LED sequencing loop:

```
*** Check for timeout
banksel time_cnt
incf    time_cnt,f    ; increment time count
btfsc   STATUS,Z
incf    time_cnt+1,f
```

Finally, we need to compare this incremented count with the timeout value, and enter standby mode if the timeout has been reached.

To do that “properly”, we’d perform a full 16-bit compare, as we did in the C version, where it was trivially easy to do. However, in assembly language programming it’s common to look for shortcuts.

In this case, stepping back and thinking about what we’re trying to achieve tells us that the timeout doesn’t have to be exact. A ten minute timeout is fine for testing, but that’s a bit short for our final product, where kids are likely to want to be able to play with their traffic lights for more than ten minutes. A timeout of about an hour is more reasonable. And if we’re aiming for “about an hour”, a few minutes more or less won’t make any really difference.

This means that we don’t really need the accuracy of a full 16-bit comparison. Instead, we can get away with comparing only the most significant bytes:

```
movlw   TIMEOUT*60/256 ; if timeout reached
xorwf   time_cnt+1,w   ; (high byte comparison only)
btfsc   STATUS,Z
goto    standby       ; enter standby mode
```

If `TIMEOUT = 10`, ‘`TIMEOUT*60/256`’ evaluates (using integer division) to 2, so our traffic lights will go into standby mode after $2 \times 256 = 512$ seconds = 8.5 minutes – reasonably close to the 10 minutes we were aiming for.

For a “1 hour” timeout, `TIMEOUT = 60` and ‘`TIMEOUT*60/256`’ evaluates to 14, so our actual timeout will be $14 \times 256 = 3584$ seconds = 59.7 minutes – which surely qualifies as “about an hour”.

Step 4: Manual operation

One of our requirements was to be able to control the traffic lights manually, by pressing a button to advance the sequence from green to amber to red then back to green.

Instead of jumping right in and adding a “manual mode” to our existing design, it’s easier to develop the manual version separately. Then, as a final step, we’ll bring the automatic and manual modes together into a single design. In that way, we’re still only adding one extra feature at a time: manual mode first, and then the ability to switch between modes.

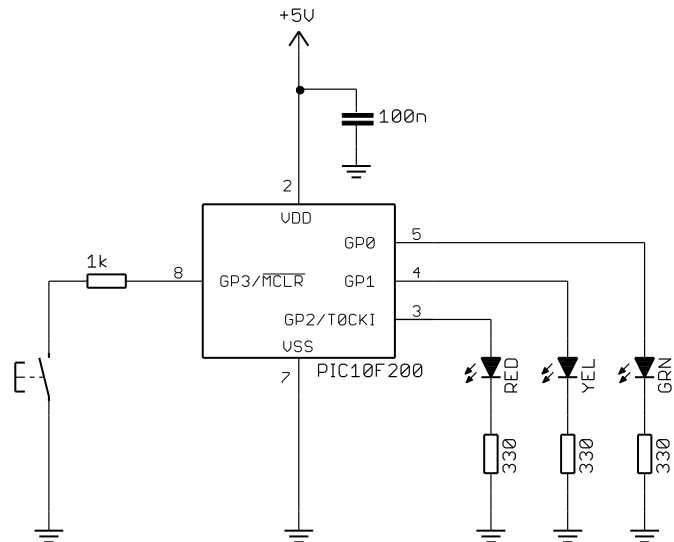
For manual operation we only need our three coloured LEDs and a pushbutton switch, so we can continue to use the circuit from step 2, as shown on the right, for now.

We'll use the pushbutton to change the light to the next in sequence.

If we also want to use the pushbutton to enter sleep mode, as we did in steps 2 and 3, we would have to implement a system such as holding the button down for a couple of seconds to turn the lights off.

However, since we've already developed a means to power off the lights in automatic mode, and we intend to make it quick and easy to switch between modes, we don't really need to add an "enter standby" function for manual mode. If the user is using the lights in manual mode, he or she can easily flick them over the automatic mode and then press the button to power them off. So we won't bother with adding an "enter standby" function for manual mode.

Of course, you may disagree with that as a design decision, in which case you can extend the software to include it – that's the beauty of programmable systems!



Conceptually the program is very simple – we could express it in pseudo-code as:

```

Initialisation:
    configure LED pins as outputs
    enable internal pull-ups
    start with only green LED on

Main loop:
do forever
    // light each LED in sequence on button press
    wait for button press
    turn off all LEDs          // change to yellow
    turn on yellow
    wait for debounced button release

    wait for button press
    turn off all LEDs          // change to red
    turn on red
    wait for debounced button release

    wait for button press
    turn off all LEDs          // change to green
    turn on green
    wait for debounced button release
end
  
```

Note that the next LED is lit immediately after the button is pressed, instead of waiting for the button to be released first. The traffic lights will feel more responsive that way.

And of course the pushbutton should be debounced, to avoid contact bounces triggering subsequent light changes.

Note that the same block of code is repeated, with minimal variations, for each light.

That can be appropriate for a program like this, where we're only repeating the same (or similar) operation a few times, although it can become unwieldy when we try to add features to it, as we saw when we went to add the power-down capability in automatic mode.

So although the above structure is very simple, it's actually difficult to build upon, and it's better to restructure the program as follows:

```

Initialisation:
    configure LED pins as outputs
    enable internal pull-ups
    start with (only) green LED on
    initial state = green

Main loop:
do forever
    wait for button press

    // light next LED in sequence
    turn off all LEDs
    select (current state)
        green:
            next state = yellow
            turn on yellow
        yellow:
            next state = red
            turn on red
        red:
            next state = green
            turn on green

    wait for debounced button release
end

```

In this way, we only wait for the button press at the start of the main loop, and then light the next LED in the sequence, depending on which LED is currently lit.

Although in principal it's possible to read the I/O port to determine which LED is currently lit, reliably reading the state of output pins can be problematic in baseline PICs⁴. It is better to use a variable to keep track of the current state, perhaps representing "green" with the value 0, "yellow" with 1 and "red" with 2.

XC8 implementation

Again, we don't need any new PIC programming techniques to implement this step in C – it's only a matter of translating the above pseudo-code.

We'll need a variable to record the current state. Although we could use numbers to represent the various states, as mentioned above, it's clearer to declare the variable as an enumerated type:

```
enum {GREEN, YELLOW, RED} state;    // state = currently-lit LED
```

The C compiler will define GREEN, YELLOW and RED as numeric constants behind the scenes, but we don't need to know their specific values; we can simply use these symbolic values by name when working with this 'state' variable.

⁴ see the discussion of the "read-modify-write" problem in [baseline assembler lesson 2](#)

For example, within our initialisation routine we now need to set the initial state to “green”:

```
// set initial state
state = GREEN;           // initial state is green, so
G_LED = 1;               // turn on green LED
```

With the green LED initially on, the main loop begins by waiting for a button press:

```
/***/ Main loop
for (;;)
{
    // wait for button press
    while (BUTTON)           // wait until button low
        ;
}
```

We then turn off whichever LED is currently lit:

```
LEDS = 0;                 // turn off all LEDs
```

And then use the current state to select which LED to light next, updating the current state to the next in sequence:

```
switch (state)           // next LED depends on currently-lit LED
{
    case GREEN:           // if green:
        state = YELLOW;  // next state = yellow
        Y_LED = 1;       // turn on yellow LED
        break;

    case YELLOW:         // if yellow:
        state = RED;     // next state = red
        R_LED = 1;       // turn on red LED
        break;

    case RED:            // if red:
        state = GREEN;   // next state = green
        G_LED = 1;       // turn on green LED
        break;
}
```

Note that C’s ‘switch’ statement corresponds to the ‘select’ construct in the pseudo-code version, and that the use of an enumerated type for the ‘state’ variable makes this very clear and easy to read.

Finally, at the end of the main loop we wait for the pushbutton to be released, and debounce it:

```
// wait for stable button release
DbnceHi (BUTTON);
```

MPASM implementation

Again, we can apply techniques we’ve already used to implement this step in assembly language.

We need to define a variable to record the current state, and for clarity we should also define constants to represent the various states:

```
VARs  UDATA
state  res 1                ; state = currently-lit LED
      constant GREEN = 0
      constant YELLOW = 1
      constant RED = 2
```

The initialisation routine is much the same as we've used before, except that we also need to set the initial state to "green":

```

; set initial state
banksel state
movlw GREEN ; initial state is green, so
movwf state
bsf G_LED ; turn on green LED

```

We then begin the main loop by waiting for a button press:

```

main_loop
;*** Wait for button press
wait_dn btfsc BUTTON ; wait until button low
goto wait_dn

```

And then turn off whichever LED is currently lit:

```

clrf LEDES ; turn off all LEDs

```

Next we can implement our pseudo-code 'select' construct, testing the current state to determine the next in sequence, updating the state and lighting the appropriate LED:

```

; test current state, to determine next LED to light
banksel state
movlw GREEN ; if green:
xorwf state,w
btfss STATUS,Z
goto man_yellow
movlw YELLOW ; next state = yellow
movwf state
bsf Y_LED ; turn on yellow LED
goto man_red_end
man_yellow
movlw YELLOW ; if yellow:
xorwf state,w
btfss STATUS,Z
goto man_red
movlw RED ; next state = red
movwf state
bsf R_LED ; turn on red LED
goto man_red_end
man_red
movlw RED ; if red:
xorwf state,w
btfss STATUS,Z
goto man_red_end
movlw GREEN ; next state = green
movwf state
bsf G_LED ; turn on green LED
man_red_end

```

Finally, at the end of the main loop we wait for the pushbutton to be released, and debounce it:

```

;*** Wait for stable button release
DbnceHi BUTTON

```

Step 5: Manual operation with timeout

Again, to save batteries, if the user hasn't pressed the "change" button for some time, the lights should enter standby mode, waking when the button is pressed again, in the same way as in automatic mode.

This is quite easy to do, if we keep the program structure from the previous step.

Instead of simply waiting for a button press, we repeatedly poll the pushbutton over a 1 second interval, as we did in step 2:

```
// delay 1 second while polling pushbutton
repeat 1000/N times
  delay N ms
  // check for button press
  if button pressed
    light next LED in sequence
    wait for debounced button release
```

Of course, if the button is pressed, lighting the next LED and then waiting for the button to be released will add to the time – this loop will take more than 1 second to execute. But that's ok – the timeout isn't supposed to be an exact amount of time. A few seconds more or less won't make any practical difference if the traffic lights are supposed to turn themselves off after "an hour or so", or even the ten minutes that we'll use for testing.

Having polled the pushbutton for a 1 second interval, we can then increment a timeout counter and enter standby mode when the time period has elapsed, as we did in step 3. The counter should be reset whenever the pushbutton is pressed, so that it is counting time since the most recent press.

Our main loop becomes, in pseudo-code:

```
Main loop:
do forever
  // delay 1 second while polling pushbutton
  repeat 1000/N times
    delay N ms
    // check for button press
    if button pressed
      time_cnt = 0      // reset timeout counter

      // light next LED in sequence
      turn off all LEDs
      select (current state)
      green:
        next state = yellow
        turn on yellow
      yellow:
        next state = red
        turn on red
      red:
        next state = green
        turn on green
      wait for debounced button release

  // increment time count and check for timeout
  time_cnt = time_cnt+1
  if time_cnt = TIMEOUT*60      // timeout in seconds
    enter standby mode
end
```

The timeout counter must of course be zeroed as part of our initialisation routine. We also need to enable wake-up on change, so that the device can be woken from sleep following a timeout.

XC8 implementation

Implementing this step in C is mostly a matter of reusing pieces of code from the earlier steps.

The `switch` statement used to select the next LED to light from step 4 is essentially placed within the polling loop from step 2, with the timeout code from step 3 added at the end of the main loop.

To see how it all fits together, it's easiest to look at the complete C program listing:

```

/*****
*
*   Description:      Simple Traffic Lights
*                   Tutorial project 1, example 5
*
*   Lights green, yellow and red lights in sequence (manual operation),
*   advancing on each pushbutton press
*
*   Power on (wake from standby) on pushbutton press
*
*   Enters standby mode if no button press
*   during timeout period (10 mins)
*
*****/
*
*   Pin assignments:
*   GP0 = green light (LED), active high
*   GP1 = yellow light (LED), active high
*   GP2 = red light (LED), active high
*   GP3 = pushbutton switch (active low)
*
*****/

#include <xc.h>
#include <stdint.h>

#include "stdmacros-XC8.h" // DbnceHi() - debounce switch, wait for high
                          // Requires: TMR0 at 256 us/tick

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog
#pragma config MCLRE = OFF, CP = OFF, WDTE = OFF

// oscillator frequency for __delay_ms()
#define _XTAL_FREQ 4000000

// Pin assignments
#define LEDS      GPIO           // all LEDs
#define G_LED    GPIObits.GP0   // individual LEDs
#define Y_LED    GPIObits.GP1
#define R_LED    GPIObits.GP2
#define BUTTON   GPIObits.GP3   // Pushbutton (active low)

/***** CONSTANTS *****/
#define POLL_MS 50              // polling interval (in ms)
#define TIMEOUT 10              // auto-off timeout (in minutes)

/***** PROTOTYPES *****/
void standby(void);            // enter standby (low-power) mode

/***** MAIN PROGRAM *****/

```



```

void main()
{
    enum {GREEN, YELLOW, RED} state;    // state = currently-lit LED

    uint16_t    time_cnt = 0;           // timeout counter (seconds since reset)
    uint8_t     p_cnt;                   // polling loop counter

    /*** Initialisation

    // configure ports
    GPIO = 0b0000;                       // start with all LEDs off
    TRIS = 0b1000;                       // configure LED pins (GP0-2) as outputs

    // configure wake-on-change, pull-ups and timer
    OPTION = 0b00000111;                 // configure wake-up on change and Timer0:
        //0----- enable wake-up on change (/GPWU = 0)
        //-0----- enable weak pull-ups (/GPPU = 0)
        //--0----- timer mode (TOCS = 0)
        /----0--- prescaler assigned to Timer0 (PSA = 0)
        /-----111 prescale = 256 (PS = 111)
        // -> increment every 256 us
        // GP2 usable as an output

    // set initial state
    state = GREEN;                       // initial state is green, so
    G_LED = 1;                           // turn on green LED

    // wait for stable button release
    // (in case it is still bouncing following wake-up on change)
    DbnceHi(BUTTON);

    /*** Main loop
    for (;;)
    {
        // delay 1 second while polling pushbutton
        // (repeat 1000/POLL_MS times)
        for (p_cnt = 0; p_cnt < 1000/POLL_MS; p_cnt++)
        {
            __delay_ms(POLL_MS);         // polling interval

            // check for button press
            if (!BUTTON)                  // if button pressed
            {
                time_cnt = 0;             // reset timeout counter

                // light next LED in sequence
                LEDS = 0;                 // turn off all LEDs

                switch (state)            // next LED depends on current LED
                {
                    case GREEN:           // if green:
                        state = YELLOW;  // next state = yellow
                        Y_LED = 1;       // turn on yellow LED
                        break;

                    case YELLOW:          // if yellow:
                        state = RED;      // next state = red
                        R_LED = 1;       // turn on red LED
                        break;
                }
            }
        }
    }
}

```

```

        case RED:                // if red:
            state = GREEN;       // next state = green
            G_LED = 1;           // turn on green LED
            break;
    }
    // wait for stable button release
    DbnceHi (BUTTON);
}

// check for timeout
if (++time_cnt == TIMEOUT*60)
    standby();                // enter standby mode
}
}

/***** FUNCTIONS *****/

/***** Enter standby (low power) mode *****/
void standby(void)
{
    LEDES = 0;                // turn off all LEDs
    DbnceHi (BUTTON);        // wait for stable button release
    SLEEP();                 // enter sleep mode
}

```

MPASM implementation

As with the C version, implementing this step with assembly language is a simple matter of placing the LED selection code from step 4 within the polling loop from step 2 while removing the automatic sequencing code, and adding the timeout code from step 3 at the end of the main loop.

Here is the complete program listing, so that you can see how this all fits together:

```

;*****
;
; Description:      Simple Traffic Lights
;                  Tutorial project 1, example 5
;
; Lights green, yellow and red lights in sequence
; (timing defined by program constants)
;
; Power on (wake from standby) on pushbutton press
;
; Enters standby mode on pushbutton press,
; or if no button press during timeout period (10 mins)
;
;*****
;
; Pin assignments:
; GP0 = green light (LED), active high
; GP1 = yellow light (LED), active high
; GP2 = red light (LED), active high
; GP3 = pushbutton switch (active low)
;
;*****

list          p=10F200
#include      <p10F200.inc>

```

```

errorlevel -312 ; no "page or bank selection not needed" messages

#include <stdmacros-base.inc> ; DbnceHi - debounce sw, wait for high
                                ; (requires TMR0 running at 256 us/tick)
                                ; DelayMS - delay in milliseconds
                                ; (calls delay10)
EXTERN delay10_R ; W x 10ms delay

radix dec

;***** CONFIGURATION
                                ; int reset, no code protect, no watchdog
__CONFIG _MCLRE_OFF & _CP_OFF & _WDTE_OFF

; pin assignments
#define LEDS GPIO ; all LEDs
#define G_LED GPIO,0 ; individual LEDs
#define Y_LED GPIO,1
#define R_LED GPIO,2
#define BUTTON GPIO,3 ; Pushbutton (active low)

;***** CONSTANTS
constant POLL_MS = 50 ; polling interval (in ms)
constant TIMEOUT = 10 ; auto-off timeout (in minutes)

;***** VARIABLE DEFINITIONS
VARS UDATA
state res 1 ; state = currently-lit LED
constant GREEN = 0
constant YELLOW = 1
constant RED = 2
time_cnt res 2 ; timeout counter (seconds since reset)
p_cnt res 1 ; polling loop counter

;***** RC CALIBRATION
RCCAL CODE 0x0FF ; processor reset vector
res 1 ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET CODE 0x000 ; effective reset vector
movwf OSCCAL ; apply internal RC factory calibration
pagesel start
goto start ; jump to main code

;***** Subroutine vectors
delay10 ; delay W x 10 ms
pagesel delay10_R
goto delay10_R

;***** MAIN PROGRAM *****
MAIN CODE

;***** Initialisation
start
; configure port

```

```

    clrf    GPIO                ; start with all LEDs off
    movlw   b'1000'            ; configure LED pins (GP0-2) as outputs
    tris    GPIO

; configure wake-on-change, pull-ups and timer
    movlw   b'00000111'        ; configure wake-up on change and Timer0:
    ; 0-----                enable wake-up on change (/GPWU = 0)
    ; -0-----                enable weak pull-ups (/GPPU = 0)
    ; --0-----               timer mode (T0CS = 0)
    ; ----0----               prescaler assigned to Timer0 (PSA = 0)
    ; -----111              prescale = 256 (PS = 111)
    option                ; -> increment every 256 us
                        ;      GP2 usable as an output

; initialise variables
    banksel time_cnt
    clrf    time_cnt          ; time_cnt = 0
    clrf    time_cnt+1

; set initial state
    banksel state
    movlw   GREEN            ; initial state is green, so
    movwf   state
    bsf     G_LED            ; turn on green LED

; wait for stable button release
; (in case it is still bouncing following wake-up on change)
    DbnceHi BUTTON

;***** Main loop
main_loop
    ;*** Delay 1 second while polling pushbutton
    banksel p_cnt
    movlw   1000/POLL_MS     ; loop 1s/(POLL_MS/loop) times
    movwf   p_cnt
poll_loop
    DelayMS POLL_MS         ; polling interval
    ; check for button press
    btfsc   BUTTON          ; if button pressed (low)
    goto    poll_end
    ;
    ; BUTTON PRESSED
    ;
    clrf    time_cnt        ; reset timeout counter
    clrf    time_cnt+1
    ;
    ; Light next LED in sequence
    ;
    clrf    LEDES           ; turn off all LEDs
    ;
    ; test current state, to determine next LED to light
    banksel state
    movlw   GREEN          ; if green:
    xorwf   state,w
    btfss   STATUS,Z
    goto    man_yellow
    movlw   YELLOW         ; next state = yellow
    movwf   state
    bsf     Y_LED          ; turn on yellow LED
    goto    man_red_end

```

```

man_yellow
    movlw    YELLOW            ;   if yellow:
    xorwf   state,w
    btfss   STATUS,Z
    goto    man_red
    movlw   RED                ;       next state = red
    movwf   state
    bsf     R_LED              ;       turn on red LED
    goto    man_red_end

man_red
    movlw   RED                ;   if red:
    xorwf   state,w
    btfss   STATUS,Z
    goto    man_red_end
    movlw   GREEN             ;       next state = green
    movwf   state
    bsf     G_LED            ;       turn on green LED

man_red_end
    ;
    ; Wait for stable button release
    DbnceHi BUTTON

poll_end
    decfsz  p_cnt,f
    goto    poll_loop

    ;*** Check for timeout
    banksel time_cnt
    incf    time_cnt,f        ; increment time count
    btfsc   STATUS,Z
    incf    time_cnt+1,f
    movlw   TIMEOUT*60/256   ; if timeout reached
    xorwf   time_cnt+1,w     ; (high byte comparison only)
    btfsc   STATUS,Z
    goto    standby         ;   enter standby mode

    ;*** Repeat forever
    goto    main_loop

;***** Standby (low power) mode
standby
    clrf    LEDS              ; turn off LEDs
    DbnceHi BUTTON          ; wait for stable button release
    sleep                               ; enter sleep mode

    END

```

Step 6: Bringing it all together

We've now developed, in steps 1 to 3, a set of automated traffic lights with an "on/off" pushbutton switch for entering and waking from standby mode, and a timeout feature.

We also developed, in steps 4 and 5, a set of manually operated traffic lights with a "change" pushbutton and a timeout feature.

It's time to bring these together into a single device which combines all these features, as outlined in the requirements at the start of this document.

As has been mentioned, our pushbutton can do double duty: it can be used as an “on/off” button in automatic mode, and as a “change” button in manual mode.

However, we do need to add a “mode” switch to the design, to allow the user to switch between automatic and manual modes.

There’s a problem with that – adding another switch means using another digital input pin, but we’ve already used every pin on the PIC10F200. So we’ll need to use a bigger PIC, with more pins.

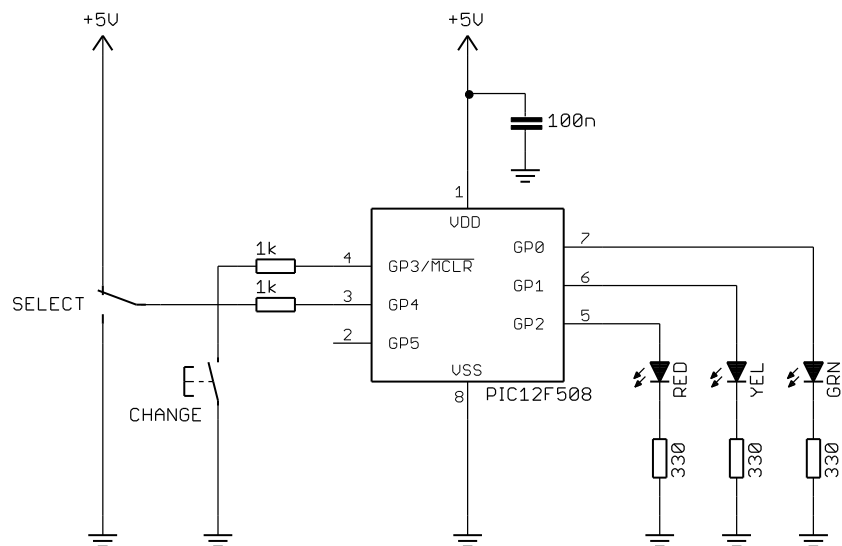
The next biggest is the 12F508. Besides gaining two more I/O pins, it has twice as much program memory (512 words) as the 10F200 and more data memory (25 bytes instead of 16 bytes), which should be more than enough because each of the two programs we’re bringing together were able to fit within a 10F200.

Although it would be possible to use a pushbutton to toggle between modes, in this application a two-position toggle or slide switch seems more appropriate. It’s simple to use – flicking the switch “up” (or “left” or “forward” etc.) could select automatic mode, while setting the switch in the other direction would select manual mode. And being a toggle or slide switch means that it’s easy to see which mode the lights are operating in (e.g. whether the switch is “up” or “down”).

We can add an SPDT toggle or slide switch to our circuit as shown on the right.

The switch is wired so that GP4 is pulled either high or low, via a 1 kΩ resistor.

Again, this resistor isn’t strictly necessary, but it protects the PIC from a situation where GP4 is inadvertently configured as an output. If the pin happened to be set to output a “high” while the switch connected it to ground (or vice-versa) the PIC could be destroyed without a resistor in place to limit the current.



Although we could in principle use a SPST switch (open in one position, closed in the other – with only two terminals) with a weak internal pull-up, as we did with the pushbutton on GP3, there’s a good reason not to do it that way. In sleep mode, the weak pull-ups must remain active – we can’t disable them before entering sleep mode, because the “change” pushbutton requires a weak pull-up for its proper operation, and we need that pushbutton to be working if it’s to be used to wake the device from sleep. But suppose the “select” switch was connected to a pin with a weak pull-up enabled? If the switch connects the pin to ground, current will flow through the pull-up to ground via the switch – draining the battery, even in standby mode.

This wouldn’t be a problem if we used a mid-range PIC, such as 12F629, where the weak pull-ups can be individually enabled. But for baseline devices such as the 12F508 it’s all (meaning GP0, GP1 and GP3) or none. That’s ok – it simply means using a pin without weak pull-ups (GP2, GP4 or GP5) for the select function, and using a double-throw (three-terminal) switch connected as shown.

Of course, as features are added to a design, you may reach a point where it makes more sense to use a different PIC architecture (or different type of microcontroller altogether). For example, you might decide

that being able to use interrupts or having individually-selectable weak pull-ups available would simplify the design to an extent that it would be worthwhile to upgrade to a mid-range PIC. That would be a perfectly valid decision, but as we'll see it's not difficult to implement this simple project with baseline devices such as the PIC12F508.

If you have the [Gooligum baseline training board](#), you can use the PIC12F509 that came with your board, instead of a 12F508⁵. Leave the board set up as before, but plug the 12F508 or 12F509 into the top section of the 14-pin IC socket marked '12F'.⁶

The training board comes with a 1 kΩ resistor, but you will need to supply your own toggle or slide switch which you can connect to GP4, VDD and GND via pins 3, 15 and 16 (respectively) on the 16-pin header.

Alternatively, you could skip the circuit in this step and instead build the circuit in the next (and final) step using the PCB and parts supplied with the [Gooligum traffic lights kit](#). As we'll see in the next step, the final version is logically the same but with different pin assignments and inverted (active low) LED operation to simplify the PCB layout. So you could read this step and then go on to the final step to implement it using the "production" hardware.

Combining the automatic and manual-mode programs, from steps 3 and 5, is straightforward.

Each includes a loop which polls the pushbutton. All we need do is, in addition to polling the pushbutton as normal, also test the select switch. If the user has flicked the switch, to select the other mode, exit the current mode and start the other one.

Although "exit the current mode" can be done via "goto" statements, these are generally frowned upon (famously "considered harmful"). It's cleaner to implement each mode as a subroutine (or function), and then to "exit the current mode" we simply exit, or return from, the current subroutine.

The initialisation routine includes only the configuration or setup operations required by both modes.

In pseudo-code we have:

Initialisation:

```
// configure hardware
configure LED pins as outputs
start with all LEDs off
enable internal pull-ups and wake-up on change

// ensure that pushbutton is not pressed
wait for BUTTON = released
debounce BUTTON

// initialise timeout count
time_cnt = 0
```

⁵ Ideally you would also specify "PIC12F509" instead of "PIC12F508" when creating your project, and, if using assembly language, modify the 'list' and '#include' directives in your code to specify "12F509" instead of "12F508". But if you don't do this, and leave your project and code configured for a 12F508, it will still work ok with a 12F509. The compiler or assembler will simply treat it as a 12F508 and won't be aware of, and therefore won't use, the 12F509's extra memory. But that's ok – it will still work just fine.

⁶ Ensure that no device is installed in the 10F socket – you can only use one PIC at a time in the training board.

The main loop then consists of continually checking the status of the select switch, and calling the appropriate mode's subroutine:

Main loop:

```
do forever
    // enter appropriate mode, depending on SELECT switch
    if SELECT = automatic
        call AutoMode
    else
        call ManualMode
end
```

Each "mode" subroutine then consists of any unique initialisation code and the main loop from each of our previous automatic and manual-mode programs, with a test within the polling loop to exit the routine if the select switch has been changed.

When either subroutine exits, we drop back into the main loop (above), which will call the subroutine corresponding to whichever mode has now been selected.

So the automatic mode subroutine becomes, in pseudo-code:

AutoMode:

```
do forever
    // light each LED in sequence, while checking for button press
    for seconds = 0 to end_cycle_time
        light appropriate LED, depending on elapsed time

        // delay 1 second while polling switches
        repeat 1000/N times
            delay N ms

            // check for button press
            if button pressed
                enter standby mode

            // check for mode change
            if SELECT = manual
                time_cnt = 0          // reset timeout counter
                exit                  // exit automatic mode

        // increment time count and check for timeout
        time_cnt = time_cnt+1
        if time_cnt = TIMEOUT*60     // timeout in seconds
            enter standby mode
    end
end
```

And the manual mode subroutine, including state initialisation, is:

ManualMode:

```
// initialise state
initial state = green
turn on (only) green LED
```



```

do forever
    // delay 1 second while polling switches
    repeat 1000/N times
        delay N ms

        // check for button press
        if button pressed
            time_cnt = 0          // reset timeout counter
            light next LED in sequence
            wait for debounced button release

        // check for mode change
        if SELECT = automatic
            time_cnt = 0          // reset timeout counter
            exit                  // exit manual mode

        // increment time count and check for timeout
        time_cnt = time_cnt+1
        if time_cnt = TIMEOUT*60 // timeout in seconds
            enter standby mode
end

```

Note that the timeout counter is reset each time we exit to select the other mode, reflecting the fact that the user interacted with the traffic lights (they changed the mode).

It would be possible to optimise this a little by resetting the timeout counter when entering each mode, instead of when exiting – meaning that the timeout counter wouldn't have to be zeroed as part of the shared initialisation code. But the intent seems clearer this way.

Another possible optimisation would be to break the “increment time count and check for timeout” code out as a separate subroutine, since the same block of timeout handling code is repeated.

But in general, whether an optimisation makes sense is an implementation decision, depending on the programming language and compiler – and sometimes the best approach isn't obvious and you have to try it both ways, to see.

XC8 implementation

Once again, to implement this step in C we can reuse much of the code from earlier steps.

We're now using a PIC12F508, which provides a range of oscillator options (unlike the 10F200, which can only use its internal RC oscillator), so we need to specify that we're using the internal RC oscillator as part of the processor configuration:

```

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC oscillator
#pragma config MCLRE = OFF, CP = OFF, WDT = OFF, OSC = IntRC

```

We also add symbolic names for the mode select switch and its possible values:

```

#define SELECT GPIObits.GP4          // mode switch:
#define SEL_auto    0                // low = auto
#define SEL_manual  1                // high = manual

```

Defining the mode values as symbols in this way will make it easier to change the user interface (perhaps the select switch should be in the “up” position for automatic mode, instead of “down”) later, without having to make changes throughout the code.

Since we'll be using automatic and manual mode functions, we need to add prototypes for them:

```

/***** PROTOTYPES *****/
void AutoMode(void);           // automatic mode
void ManualMode(void);        // manual mode
void standby(void);           // enter standby (low-power) mode

```

The main loop is simply:

```

/***/ Main loop
for (;;)
{
    // enter appropriate mode, depending on select switch
    if (SELECT == SEL_auto)
        AutoMode();
    else
        ManualMode();
}

```

Now we can take the main loop from the automatic mode program developed in step 3 and encapsulate it as a function.

At the start of the function we need to declare the variables that are only used within (“local to”) that function. So we have:

```

/***** Automatic mode *****/
void AutoMode(void)
{
    uint8_t      sec_cnt;           // seconds counter (for LED sequencing)
    uint16_t     time_cnt = 0;     // timeout counter (seconds since reset)
    uint8_t      p_cnt;           // polling loop counter
}

```

Note that the timeout counter is zeroed as part of the declaration, so it will be reset to zero every time that automatic mode is entered. This means that there is no need to zero it as part of the shared initialisation code, nor does it need to be reset when exiting manual mode; there is no need to access this ‘time_cnt’ variable from outside this function, so it can be declared as a local variable.

When we come to the manual mode function, you'll see that it also has timeout and polling loop counters, so you may think that we could save data memory by declaring them as global variables, accessed by both functions. Actually, that's not the case. The compiler allocates storage for non-static local (“auto”) variables like these from a shared memory pool, on demand – storage is only allocated to a function's local variables while that function is running (unless they are declared to be “static”), and can be reused by another function's variables when needed.

The upshot of this is that we don't waste any data memory by declaring variables within a function like this, even when there are variables with the same name in another function. An advantage of doing so is that the function is then self-contained, making the program more maintainable and code re-use easier.

The previous “main loop” then becomes a “for (;;)” loop within this function. Most of the code is the same as before, so we we'll only list the comments for the unchanged sections here (there will be a full listing at the end of the next step), except that the polling loop now also reads the mode select switch:

```

for (;;)
{
    // light each LED in sequence
    for (sec_cnt = 0; sec_cnt < R_END; sec_cnt++)
    {

```

```

// light appropriate LED, depending on elapsed time

// delay 1 second while polling pushbutton
// (repeat 1000/POLL_MS times)
for (p_cnt = 0; p_cnt < 1000/POLL_MS; p_cnt++)
{
    __delay_ms(POLL_MS);    // polling interval

    // check for button press

    // check for mode change
    if (SELECT == SEL_manual)
        return;            // exit automatic mode
}

// check for timeout
}
}

```

The approach for the manual mode function is the same, encapsulating the main loop from the program developed in step 5 as a function, except that we also need to include initialisation code to set the initial state to “green”. So the manual mode function begins, including variable definitions, with:

```

/***** Manual mode *****/
void ManualMode(void)
{
    enum {GREEN, YELLOW, RED} state;    // state = currently-lit LED

    uint16_t    time_cnt = 0;           // timeout counter (seconds since reset)
    uint8_t     p_cnt;                  // polling loop counter

    // set initial state
    state = GREEN;                     // initial state is green, so
    LEDS = 0;
    G_LED = 1;                          // turn on green LED (only)
}

```

As in automatic mode, the previous “main loop” becomes a “for (;;)” loop within this function, and again most of the code is the same as before, so we we’ll only list the comments for the unchanged sections here, except for the extra code within the polling loop which reads the mode select switch:

```

for (;;)
{
    // delay 1 second while polling pushbutton
    // (repeat 1000/POLL_MS times)
    for (p_cnt = 0; p_cnt < 1000/POLL_MS; p_cnt++)
    {
        __delay_ms(POLL_MS);    // polling interval

        // check for button press

        // check for mode change
        if (SELECT == SEL_auto)
            return;            // exit manual mode
    }

    // check for timeout
}
}

```

MPASM implementation

Once again, we can reuse much of our earlier code when implementing this step with assembly language.

Now that we're using a PIC12F508, we need to specify it, using the `list` and `#include` directives at the start of the program:

```
list          p=12F508
#include      <p12F508.inc>
```

We also now need to specify that we're using the internal oscillator, as part of the processor configuration:

```
***** CONFIGURATION
; int reset, no code protect, no watchdog, int RC oscillator
__CONFIG    _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC
```

We should define a symbolic name for the mode select switch:

```
#define SELECT      GPIO,4          ; mode switch (low = auto, high = manual)
```

However (unlike the C version), it does not make sense to define symbols for the switch values that represent automatic and manual modes, because we'll be using bit-test (`btfs` and `btfs`) instructions to read and react to the select switch input – there is no opportunity to explicitly compare the input value with a constant, as we can do so easily in C.

One possible approach to abstracting these types of input pin tests, to make it easier to change the user interface later (so that a low input means “automatic” instead of “manual”) is to encapsulate the bit test instructions as macros, using the macros in your code in place of the bit test instructions, and updating the macro definitions if the pin assignments or meanings change. In fact it's possible to take that to a level where the code no longer looks much like assembly language – but then it's arguable that you then might as well have been using C. So, we'll keep it simple here.

The variable definitions combine those from both the automatic and manual mode programs:

```
***** VARIABLE DEFINITIONS
VARS          UDATA
sec_cnt       res 1          ; seconds counter (for LED sequencing)
state         res 1          ; state = currently-lit LED
              constant GREEN = 0
              constant YELLOW = 1
              constant RED = 2
time_cnt      res 2          ; timeout counter (seconds since reset)
p_cnt         res 1          ; polling loop counter
```

Since we'll be using automatic and manual mode subroutines, we should add them to our jump table:

```
***** Subroutine vectors
delay10                               ; delay W x 10 ms
    pagesel delay10_R
    goto    delay10_R

AutoMode                               ; automatic mode
    pagesel AutoMode_R
    goto    AutoMode_R

ManualMode                             ; manual mode
    pagesel ManualMode_R
    goto    ManualMode_R
```

Recall that the main loop will consist of continually reading the select switch input, and then running the appropriate subroutine.

In pseudo-code, we wrote this as:

```
do forever
    // enter appropriate mode, depending on SELECT switch
    if SELECT = automatic
        call AutoMode
    else
        call ManualMode
end
```

We could translate that directly into assembly language, but given the need for `pagesel` directives when calling subroutines and also for the `'goto'` instructions needed to jump around the code blocks within the "if/else" structure (see [baseline assembler lesson 3](#))⁷, it gets a bit messy.

It's easier to implement it in assembly language if we note that, after exiting automatic mode, we will always want to enter manual mode next (because the select switch must have changed).

So it is actually quite ok to drop the "else" and simplify it to:

```
do forever
    // enter appropriate mode, depending on SELECT switch
    if SELECT = automatic
        call AutoMode
    call ManualMode
end
```

And that translates quite neatly into assembly language as:

```
;***** Main loop
main_loop
    ; enter appropriate mode, depending on select switch
    pagesel AutoMode
    btfss    SELECT          ; if automatic (low)
    call    AutoMode
    call    ManualMode      ; else (or then) enter manual mode

    ;** Repeat forever
    pagesel main_loop
    goto   main_loop
```

The automatic mode subroutine consists of the main loop from the automatic mode program developed in step 3, with the timeout counter being reset at the start of the subroutine:

```
;***** SUBROUTINES *****
SUBS    CODE

;***** Automatic mode
AutoMode_R
    ; initialise variables
    banksel time_cnt
    clrf    time_cnt        ; time_cnt = 0
    clrf    time_cnt+1
```

⁷ not strictly needed on a 12F508, which has only a single page of memory, but good practice to include in case the program is ever migrated to a 12F509 or any other baseline PIC with more than memory page

Most of the loop code is the same as before, so we we'll only list the comments for most of the unchanged sections here (there will be a full listing at the end of the next step), except that the polling loop now also reads the mode switch, and exits (returns from) the subroutine if manual mode has been selected:

```

auto_start
    ; initialise seconds count (used to light each LED in sequence)
    banksel sec_cnt          ; sec_cnt = 0
    clrf    sec_cnt

auto_loop
    ;*** Light appropriate LED, depending on elapsed time

    ;*** Delay 1 second while polling pushbutton
    banksel p_cnt
    movlw   1000/POLL_MS    ; loop 1s/(POLL_MS/loop) times
    movwf   p_cnt
auto_poll_loop
    DelayMS POLL_MS        ; polling interval
    ; check for button press
    btfss   BUTTON         ; if button down (low)
    goto    standby        ; enter standby mode
    ; check for mode change
    btfsc   SELECT         ; if manual mode selected (high)
    retlw   0              ; exit automatic mode
    ; end polling loop
    decfsz  p_cnt,f
    goto    auto_poll_loop

    ;*** Check for timeout

    ;*** End seconds count loop
    banksel sec_cnt
    incf    sec_cnt,w      ; sec_cnt = sec_cnt+1
    movwf   sec_cnt
    xorlw   R_END         ; loop until sec_cnt = R_END
    btfss   STATUS,Z
    goto    auto_loop

    ;*** Repeat (until mode change or timeout)
    goto    auto_start

```

The manual mode subroutine consists of the main loop from the program developed in step 5, plus initialisation code which resets the timeout counter and sets the initial state to “green”.

So the manual mode subroutine begins with:

```

;***** Manual mode
ManualMode_R
    ; initialise variables
    banksel time_cnt
    clrf    time_cnt      ; time_cnt = 0
    clrf    time_cnt+1

    ; set initial state
    banksel state
    movlw   GREEN         ; initial state is green, so
    movwf   state
    clrf    LEDES
    bsf     G_LED         ; turn on green LED (only)

```


We also need to replace idealised components, such as the switches shown in the earlier circuit diagrams, with real devices. In this case, the “select” switch becomes a PCB-mounted slide switch, with additional “terminals” (shown as ‘B’ and ‘B1’) having no electrical connection but used for mounting. Similarly, the “change” switch becomes a PCB-mounted pushbutton with four terminals connected in pairs, as shown.

Sometimes, when “productionising” a design, components are added to make the design more robust, such as the diode used to protect the PIC from reverse or over-voltage. But you may find that some parts can be omitted safely. For example, as long as we’re certain that our code has been fully debugged and that there is no possibility that the switch input pins will be programmed as outputs, it’s ok to drop the resistors that we had previously placed between each switch and input pin. For a truly robust design, you would never do this, but for a cheap toy where you’re certain that the program is production-ready and won’t be programmed in-circuit, it’s ok to remove these resistors.

Normally, when driving LEDs, while it’s ok to drive a number of LEDs in series (where the current in each LED will be the same), each “string” of LEDs should have its own current-limiting resistor, because LEDs in parallel are not guaranteed to share current evenly. However, our traffic lights will only ever have one LED lit at once. And since only one LED will ever be lit at once, it is ok for them to share a single current-limiting resistor, as shown.

This resistor has been reduced to $180\ \Omega$ to increase the LED current; with a power supply of $5.3\ \text{V}$ ($6.0\ \text{V}$ supplied by the batteries minus a drop of $0.7\ \text{V}$ across the diode) and an LED forward voltage of $1.8\ \text{V}$, the diode current will be $(5.3\ \text{V} - 1.8\ \text{V}) / 180\ \Omega = 19.4\ \text{mA}$. That’s well within the $25\ \text{mA}$ that each pin can source or supply, while lighting the LEDs brightly.

It’s good practice to tie any unused inputs high or low, instead of leaving them to “float”, especially for CMOS inputs (such as those on the PIC12F508), where floating inputs can lead to high current draw by the CMOS input circuitry. So you may wonder why the GP3 input is left disconnected. In fact, it’s not. Our program code enables weak pull-ups on all input pins with that facility, which on the 12F508 includes GP3. So in fact the GP3 input isn’t left floating; it’s pulled to VDD internally.

One of the advantages of working with microcontrollers is that it’s often possible to simplify the PCB design by remapping the I/O pins, making it easier to layout tracks and, for a simple circuit like this one, avoid the need for a double-sided board or links. Of course, we do have some constraints: GP3 cannot be used as an output, and weak pull-ups and wake-up on change are only available on GP0, GP1 and GP3. However, by rearranging the pin assignments as shown, it was possible to design a simple, single-sided PCB with the switches and battery and LED connectors in appropriate locations.

But note that another, more significant change was also made to the design to make the PCB layout process easier: the LEDs are now connected as active-low devices (the pin being pulled low to turn on the LED connected to it), instead of the active-high approach we’ve used so far.

Although active-high is more intuitive (“make the pin ‘high’ to light the LED” is easy to grasp), the pins on a PIC12F508 can sink as much current as they can source. There is no electrical reason to choose one approach over the other; both are perfectly valid. And if having active-low LEDs simplifies the PCB layout, why not make that change to the design? It’s easy to modify the software for active-low operation; simply set each output pin low, instead of high, to turn on that pin’s LED.

When designing, you should consider both active-high and active-low operation to be valid, and be prepared to select either depending on electrical, component, or layout considerations.

This final version of the circuit is the same as that used in the [Gooligum traffic lights kit](#). To build it, you could supply and breadboard the parts yourself (note that it's not possible to use the LEDs on the [Gooligum baseline training board](#) directly, because they are set up for active-high operation), or you could purchase the kit, or just the PCB, from [www.gooligum.com.au](#).

To modify the previous program to with the production hardware, we only need to update the pin assignments and invert the operation of all the “turn off/on LED” code, for active-low operation.

But since this is now the final version, we should also increase the timeout from 10 minutes to a more realistic 60 minutes – or whatever you think is appropriate.

XC8 implementation

The program is the same as that in the previous step, except for the changes mentioned above.

The pin assignments become:

```
// Pin assignments
#define LEDS      GPIO           // all LEDs
#define G_LED    GPIObits.GP0   // individual LEDs
#define Y_LED    GPIObits.GP4
#define R_LED    GPIObits.GP5
#define BUTTON   GPIObits.GP1   // pushbutton (active low)
#define SELECT   GPIObits.GP2   // mode switch:
#define SEL_auto 0              // low = auto
#define SEL_manual 1           // high = manual
```

And the timeout value is changed to 60 minutes:

```
#define TIMEOUT 60              // auto-off timeout (in minutes)
```

We need to change the initialisation routine to reflect the fact that a different set of pins (GP0, GP4 and GP5) are now outputs, and that pins must now be set ‘high’ to turn off the LEDs:

```
// configure ports
GPIO = 0b111111;              // start with all LEDs off
TRIS = 0b0011110;            // configure LED pins (GP0,4,5) as outputs
```

And then throughout the rest of the code, to turn off all the LEDs we use:

```
LEDS = 0b111111;             // turn off all LEDs
```

Note that we could instead use:

```
LEDS = 0b110001;            // turn off all LEDs
```

because the LEDs are only connected to GP0, GP4 and GP5. But that would be harder to maintain – we might reassign the pins again someday. By setting every output pin high (this statement won't affect any pins configured as inputs), we are sure to turn off every connected (active-low) LED.

Finally to turn on a single LED we have, for example:

```
G_LED = 0;                  // turn on green LED
```

Here is the complete, and final, C program listing:

```

/*****
*
*   Description:      Simple Traffic Lights
*                   Tutorial project 1, example 7
*                   (final production version)
*
*   Automatic or manual operation, selected by slide switch
*
*   Automatic mode:
*       Sequence G->Y->R->G based on preset times
*       Power down (standby) on button press
*
*   Manual mode:
*       G->Y, Y->R, R->G transitions on button press
*
*   Mode can be changed through select switch at any time
*
*   Power on (wake from standby) on button press
*
*   Power off (standby) on button press in automatic modes,
*   or if no button press or switch change
*   during timeout period (60 mins)
*
*****/
*
*   Pin assignments:
*       GP0 = green  light (LED), active low
*       GP4 = yellow light (LED), active low
*       GP5 = red    light (LED), active low
*       GP1 = pushbutton switch (active low)
*       GP2 = slide switch (low = auto, high = manual mode)
*
*****/

#include <xc.h>
#include <stdint.h>

#include "stdmacros-XC8.h" // DbnceHi() - debounce switch, wait for high
                          // Requires: TMR0 at 256 us/tick

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC oscillator
#pragma config MCLRE = OFF, CP = OFF, WDT = OFF, OSC = IntRC

// oscillator frequency for __delay_ms()
#define _XTAL_FREQ 4000000

// Pin assignments
#define LEDS      GPIO           // all LEDs
#define G_LED    GPIObits.GP0    // individual LEDs
#define Y_LED    GPIObits.GP4
#define R_LED    GPIObits.GP5
#define BUTTON   GPIObits.GP1    // pushbutton (active low)
#define SELECT   GPIObits.GP2    // mode switch:
#define SEL_auto 0                // low = auto
#define SEL_manual 1             // high = manual

/***** CONSTANTS *****/

```

```

#define G_TIME 12 // time (secs) each colour is on for
#define Y_TIME 3
#define R_TIME 10

#define G_START 0 // seconds into cycle to turn on each LED
#define Y_START G_TIME
#define R_START Y_START + Y_TIME

#define R_END R_START + R_TIME // total cycle length

#define POLL_MS 50 // polling interval (in ms)
#define TIMEOUT 60 // auto-off timeout (in minutes)

/***** PROTOTYPES *****/
void AutoMode(void); // automatic mode
void ManualMode(void); // manual mode
void standby(void); // enter standby (low-power) mode

/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation

    // configure ports
    GPIO = 0b1111111; // start with all LEDs off
    TRIS = 0b0011110; // configure LED pins (GP0,4,5) as outputs

    // configure wake-on-change, pull-ups and timer
    OPTION = 0b00000111; // configure wake-up on change and Timer0:
        //0----- enable wake-up on change (/GPWU = 0)
        //-0----- enable weak pull-ups (/GPPU = 0)
        //--0----- timer mode (T0CS = 0)
        /----0--- prescaler assigned to Timer0 (PSA = 0)
        /-----111 prescale = 256 (PS = 111)
        // -> increment every 256 us
        // GP2 usable as an output

    // wait for stable button release
    // (in case it is still bouncing following wake-up on change)
    DbnceHi(BUTTON);

    /*** Main loop
    for (;;)
    {
        // enter appropriate mode, depending on select switch
        if (SELECT == SEL_auto)
            AutoMode();
        else
            ManualMode();
    }
}

/***** FUNCTIONS *****/

/***** Automatic mode *****/
void AutoMode(void)
{

```

```

uint8_t      sec_cnt;           // seconds counter (for LED sequencing)
uint16_t     time_cnt = 0;     // timeout counter (seconds since reset)
uint8_t      p_cnt;           // polling loop counter

for (;;)
{
    // light each LED in sequence
    for (sec_cnt = 0; sec_cnt < R_END; sec_cnt++)
    {
        // light appropriate LED, depending on elapsed time
        if (sec_cnt == G_START)
        {
            LEDES = 0b111111;    // turn off all LEDs
            G_LED = 0;           // turn on green LED
        }
        if (sec_cnt == Y_START)
        {
            LEDES = 0b111111;    // turn off all LEDs
            Y_LED = 0;           // turn on yellow LED
        }
        if (sec_cnt == R_START)
        {
            LEDES = 0b111111;    // turn off all LEDs
            R_LED = 0;           // turn on red LED
        }

        // delay 1 second while polling pushbutton
        // (repeat 1000/POLL_MS times)
        for (p_cnt = 0; p_cnt < 1000/POLL_MS; p_cnt++)
        {
            __delay_ms(POLL_MS); // polling interval

            // check for button press
            if (!BUTTON)
                standby();        // enter standby mode

            // check for mode change
            if (SELECT == SEL_manual)
                return;           // exit automatic mode
        }

        // check for timeout
        if (++time_cnt == TIMEOUT*60)
            standby();           // enter standby mode
    }
}

/***** Manual mode *****/
void ManualMode(void)
{
    enum {GREEN, YELLOW, RED} state; // state = currently-lit LED

    uint16_t     time_cnt = 0;     // timeout counter (seconds since reset)
    uint8_t      p_cnt;           // polling loop counter

    // set initial state
    state = GREEN;                // initial state is green, so
    LEDES = 0b111111;
    G_LED = 0;                    // turn on green LED (only)
}

```

```

for (;;)
{
    // delay 1 second while polling pushbutton
    // (repeat 1000/POLL_MS times)
    for (p_cnt = 0; p_cnt < 1000/POLL_MS; p_cnt++)
    {
        __delay_ms(POLL_MS);    // polling interval

        // check for button press
        if (!BUTTON)           // if button pressed
        {
            time_cnt = 0;      // reset timeout counter

            // light next LED in sequence
            LEDES = 0b111111;  // turn off all LEDs

            switch (state)     // next LED depends on current LED
            {
                case GREEN:    // if green:
                    state = YELLOW; // next state = yellow
                    Y_LED = 0;    // turn on yellow LED
                    break;

                case YELLOW:   // if yellow:
                    state = RED;  // next state = red
                    R_LED = 0;    // turn on red LED
                    break;

                case RED:      // if red:
                    state = GREEN; // next state = green
                    G_LED = 0;    // turn on green LED
                    break;
            }
            // wait for stable button release
            DbnceHi(BUTTON);
        }

        // check for mode change
        if (SELECT == SEL_auto)
            return;           // exit manual mode
    }

    // check for timeout
    if (++time_cnt == TIMEOUT*60)
        standby();          // enter standby mode
}

/***** Enter standby (low power) mode *****/
void standby(void)
{
    LEDES = 0b111111;    // turn off all LEDs
    DbnceHi(BUTTON);    // wait for stable button release
    SLEEP();            // enter sleep mode
}

```

MPASM implementation

As with the C version, the assembly language program is the same as in the previous step, except for changes to the pin assignments, timeout, and inverted (active-low) outputs.

The pin assignments become:

```
; pin assignments
#define LEADS          GPIO          ; all LEDs
#define G_LED         GPIO,0        ; individual LEDs
#define Y_LED         GPIO,4
#define R_LED         GPIO,5
#define BUTTON        GPIO,1        ; pushbutton (active low)
#define SELECT        GPIO,2        ; mode switch (low = auto, high = manual)
```

And the timeout value is changed to 60 minutes:

```
constant TIMEOUT = 60                ; auto-off timeout (in minutes)
```

The initialisation routine is changed to reflect the fact that a different set of pins (GP0, GP4 and GP5) are now outputs, and that pins must now be set 'high' to turn off the LEDs:

```
; configure port
movlw  b'111111'
movwf  GPIO                ; start with all LEDs off
movlw  b'001110'          ; configure LED pins (GP0,4,5) as outputs
tris  GPIO
```

Throughout the rest of the code, to turn off all the LEDs we use:

```
movlw  b'111111'          ; turn off all LEDs
movwf  LEADS
```

Since we only have LEDs on GP0, GP4 and GP5, we could instead use:

```
movlw  b'110001'          ; turn off all LEDs
movwf  LEADS
```

But as noted for the C version, it would be harder to maintain.

Finally to turn on a single LED we have, for example:

```
bcf    G_LED              ; turn on green LED
```

This means that, to turn off all the LEDs except one, we have for example:

```
movlw  b'111111'          ; turn off all LEDs
movwf  LEADS
bcf    G_LED              ; turn on green LED
```

That's ok but a bit unwieldy. You might prefer to define symbols to represent the port bit numbers corresponding to each LED, for example:

```
; pin assignments
#define LEADS          GPIO          ; all LEDs
#define G_LED         GPIO,0        ; individual LEDs
constant nG_LED = 0      ; (green on pin 0)
#define Y_LED         GPIO,4
constant nY_LED = 4     ; (yellow on pin 4)
```

etc.

You could then write, to turn off all the LEDs except green, for example:

```
    movlw    ~(1<<nG_LED)    ; turn on green LED (only)
    movwf    LEDS
```

That saves an instruction and is just as maintainable, but whether it's clearer is really a matter of personal preference.

Here is the complete, and final, assembly language program listing:

```
;*****
;   Description:      Simple Traffic Lights                               *
;                   Tutorial project 1, example 7                       *
;                   (final production version)                           *
;                   *                                                   *
;   Automatic or manual operation, selected by slide switch            *
;                   *                                                   *
;   Automatic mode:                                                 *
;   Sequence G->Y->R->G based on preset times                           *
;   Power down (standby) on button press                               *
;                   *                                                   *
;   Manual mode:                                                    *
;   G->Y, Y->R, R->G transitions on button press                         *
;                   *                                                   *
;   Mode can be changed through select switch at any time            *
;                   *                                                   *
;   Power on (wake from standby) on button press                       *
;                   *                                                   *
;   Power off (standby) on button press in automatic modes,           *
;   or if no button press or switch change                            *
;   during timeout period (60 mins)                                   *
;                   *
;*****
;
;   Pin assignments:                                                 *
;   GP0 = green  light (LED), active low                               *
;   GP4 = yellow light (LED), active low                               *
;   GP5 = red    light (LED), active low                               *
;   GP1 = pushbutton switch (active low)                               *
;   GP2 = slide switch (low = auto, high = manual mode)               *
;                   *
;*****

list          p=12F508
#include       <p12F508.inc>

errorlevel   -312      ; no "page or bank selection not needed" messages

#include       <stdmacros-base.inc>  ; DbnceHi - debounce sw, wait for high
                                           ; (requires TMR0 running at 256 us/tick)
                                           ; DelayMS - delay in milliseconds
                                           ; (calls delay10)
EXTERN        delay10_R              ; W x 10ms delay

radix         dec

;***** CONFIGURATION
                ; int reset, no code protect, no watchdog, int RC oscillator
__CONFIG      _MCLRE_OFF & _CP_OFF & _WDT_OFF & _Intrc_OSC
```

```

; pin assignments
#define LEADS          GPIO          ; all LEDs
#define G_LED         GPIO,0        ; individual LEDs
#define Y_LED         GPIO,4
#define R_LED         GPIO,5
#define BUTTON        GPIO,1        ; pushbutton (active low)
#define SELECT        GPIO,2        ; mode switch (low = auto, high = manual)

;***** CONSTANTS
constant G_TIME = 12                ; time (seconds) each colour is on for
constant Y_TIME = 3
constant R_TIME = 10

constant G_START = 0                ; seconds into cycle to turn on each LED
constant Y_START = G_TIME
constant R_START = Y_START + Y_TIME

constant R_END = R_START + R_TIME   ; total cycle length

constant POLL_MS = 50               ; polling interval (in ms)
constant TIMEOUT = 60               ; auto-off timeout (in minutes)

;***** VARIABLE DEFINITIONS
VARS          UDATA
sec_cnt       res 1                  ; seconds counter (for LED sequencing)
state         res 1                  ; state = currently-lit LED
              constant GREEN = 0
              constant YELLOW = 1
              constant RED = 2
time_cnt      res 2                  ; timeout counter (seconds since reset)
p_cnt         res 1                  ; polling loop counter

;***** RC CALIBRATION
RCCAL        CODE    0x0FF          ; processor reset vector
              res 1                  ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****
RESET        CODE    0x000          ; effective reset vector
              movwf   OSCCAL         ; apply internal RC factory calibration
              pagesel start
              goto    start         ; jump to main code

;***** Subroutine vectors
delay10
              pagesel delay10_R
              goto    delay10_R

AutoMode
              ; automatic mode
              pagesel AutoMode_R
              goto    AutoMode_R

ManualMode
              ; manual mode
              pagesel ManualMode_R
              goto    ManualMode_R

;***** MAIN PROGRAM *****

```



```

MAIN    CODE

;***** Initialisation
start
    ; configure port
    movlw    b'1111111'
    movwf    GPIO
    movlw    b'0011110'
    tris     GPIO
                ; start with all LEDs off
                ; configure LED pins (GP0,4,5) as outputs

    ; configure wake-on-change, pull-ups and timer
    movlw    b'00000111'
                ; configure wake-up on change and Timer0:
                ; 0----- enable wake-up on change (/GPWU = 0)
                ; -0----- enable weak pull-ups (/GPPU = 0)
                ; --0----- timer mode (T0CS = 0)
                ; ----0---- prescaler assigned to Timer0 (PSA = 0)
                ; -----111 prescale = 256 (PS = 111)
    option
                ; -> increment every 256 us
                ; GP2 usable as an output

    ; wait for stable button release
    ; (in case it is still bouncing following wake-up on change)
    DbnceHi BUTTON

;***** Main loop
main_loop
    ; enter appropriate mode, depending on select switch
    pagesel AutoMode
    btfss    SELECT
                ; if automatic (low)
    call     AutoMode
    call     ManualMode
                ; else (or then) enter manual mode

    ;*** Repeat forever
    pagesel main_loop
    goto    main_loop

;***** Standby (low power) mode
standby
    movlw    b'1111111'
                ; turn off all LEDs
    movwf    LEDS
    DbnceHi BUTTON
                ; wait for stable button release
    sleep
                ; enter sleep mode

;***** SUBROUTINES *****
SUBS     CODE

;***** Automatic mode
AutoMode_R
    ; initialise variables
    banksel time_cnt
    clrf     time_cnt
                ; time_cnt = 0
    clrf     time_cnt+1

auto_start
    ; initialise seconds count (used to light each LED in sequence)
    banksel sec_cnt
    clrf     sec_cnt
                ; sec_cnt = 0

```

```

auto_loop
;*** Light appropriate LED, depending on elapsed time
banksel sec_cnt
movf    sec_cnt,w      ; if sec_cnt = G_START
xorlw   G_START
btfss  STATUS,Z
goto   auto_yellow
movlw  b'111111'      ; turn off all LEDs
movwf  LEDS
bcf    G_LED          ; turn on green LED
auto_yellow
movf    sec_cnt,w      ; if sec_cnt = Y_START
xorlw   Y_START
btfss  STATUS,Z
goto   auto_red
movlw  b'111111'      ; turn off all LEDs
movwf  LEDS
bcf    Y_LED          ; turn on yellow LED
auto_red
movf    sec_cnt,w      ; if sec_cnt = R_START
xorlw   R_START
btfss  STATUS,Z
goto   auto_red_end
movlw  b'111111'      ; turn off all LEDs
movwf  LEDS
bcf    R_LED          ; turn on red LED
auto_red_end

;*** Delay 1 second while polling pushbutton
banksel p_cnt
movlw  1000/POLL_MS    ; loop 1s/(POLL_MS/loop) times
movwf  p_cnt
auto_poll_loop
DelayMS POLL_MS        ; polling interval
; check for button press
btfss  BUTTON          ; if button down (low)
goto   standby         ; enter standby mode
; check for mode change
btfsc  SELECT          ; if manual mode selected (high)
retlw  0                ; exit automatic mode
; end polling loop
decfsz p_cnt,f
goto   auto_poll_loop

;*** Check for timeout
banksel time_cnt
incf   time_cnt,f      ; increment time count
btfsc  STATUS,Z
incf   time_cnt+1,f
movlw  TIMEOUT*60/256 ; if timeout reached
xorwf  time_cnt+1,w    ; (high byte comparison only)
btfsc  STATUS,Z
goto   standby         ; enter standby mode

;*** End seconds count loop
banksel sec_cnt
incf   sec_cnt,w      ; sec_cnt = sec_cnt+1
movwf  sec_cnt
xorlw  R_END          ; loop until sec_cnt = R_END
btfss  STATUS,Z
goto   auto_loop

```

```

    ;*** Repeat (until mode change or timeout)
    goto    auto_start

;***** Manual mode
ManualMode_R
    ; initialise variables
    banksel time_cnt
    clrf    time_cnt        ; time_cnt = 0
    clrf    time_cnt+1

    ; set initial state
    banksel state
    movlw   GREEN          ; initial state is green, so
    movwf   state
    movlw   b'111111'      ; turn off all LEDs
    movwf   LEDS
    bcf     G_LED          ; turn on green LED (only)

man_start
    ;*** Delay 1 second while polling pushbutton
    banksel p_cnt
    movlw   1000/POLL_MS   ; loop 1s/(POLL_MS/loop) times
    movwf   p_cnt
man_poll_loop
    DelayMS POLL_MS        ; polling interval
    ; check for button press
    btfsc   BUTTON        ; if button pressed (low)
    goto    man_poll_end
    ;
    ; BUTTON PRESSED
    ;
    clrf    time_cnt        ; reset timeout counter
    clrf    time_cnt+1
    ;
    ; Light next LED in sequence
    ;
    movlw   b'111111'      ; turn off all LEDs
    movwf   LEDS
    ;
    ; test current state, to determine next LED to light
    banksel state
    movlw   GREEN          ; if green:
    xorwf   state,w
    btfss   STATUS,Z
    goto    man_yellow
    movlw   YELLOW        ; next state = yellow
    movwf   state
    bcf     Y_LED          ; turn on yellow LED
    goto    man_red_end
man_yellow
    movlw   YELLOW        ; if yellow:
    xorwf   state,w
    btfss   STATUS,Z
    goto    man_red
    movlw   RED           ; next state = red
    movwf   state
    bcf     R_LED          ; turn on red LED
    goto    man_red_end
man_red
    movlw   RED           ; if red:

```

```

        xorwf    state,w
        btfss   STATUS,Z
        goto    man_red_end
        movlw   GREEN           ;       next state = green
        movwf   state
        bcf     G_LED           ;       turn on green LED
man_red_end
        ;
        ; Wait for stable button release
        DbnceHi BUTTON

man_poll_end
        ; check for mode change
        btfss   SELECT         ; if automatic mode selected (low)
        retlw   0              ;   exit manual mode
        ; end polling loop
        decfsz  p_cnt,f
        goto    man_poll_loop

        ;*** Check for timeout
        banksel time_cnt
        incf    time_cnt,f     ; increment time count
        btfsc   STATUS,Z
        incf    time_cnt+1,f
        movlw   TIMEOUT*60/256 ; if timeout reached
        xorwf   time_cnt+1,w   ; (high byte comparison only)
        btfsc   STATUS,Z
        goto    standby       ;   enter standby mode

        ;*** Repeat (until mode change or timeout)
        goto    man_start

END

```

Conclusion

Who would have thought that toy traffic lights could be so complicated?

They're not really – but we've seen in this project that even a simple device may have a number of features, and that, when developing the device, it can make sense to start with the most basic functionality, get that working, and then add one feature at a time. And that, if your device has more than one operating mode, it can be best to implement each mode separately, and then combine them later.

We also saw, when we added pushbutton polling to the basic automatic mode, that to add an apparently simple feature can mean having to rethink our approach and restructure the program.

Finally, we saw that, when taking the step from prototype to production, it may make sense to reassign the pins allocated to various I/O functions, and perhaps even change the way that those functions work – such as the change from active-high to active-low LED operation.

And if you're an assembly language programmer, you might have noticed that the C versions of the examples are shorter and easier to follow than the assembly language versions! C programmers will of course already have known this...

Simple Minute Timer

Construction and Operation Guide

Most commercial household timers have two or three buttons which must be held down or repeatedly pressed to enter the time, which can be a tedious process, and an LCD display, which can be difficult to see in poor lighting.

This compact, hand held, battery-powered timer solves those problems by using a keypad to allow the time to simply be typed in directly, and a bright LED display that's easily visible in bright or dim light. It also has a penetrating, yet not too loud, alarm

Features

- Bright four-digit LED display, showing hours and minutes
- Duration up to 99 hours and 99 minutes
- Alarm (piezo speaker)
- Flashing seconds indicator
- Simple keypad operation
- Timer can be paused
- Battery powered (2 × AA batteries)
- Turns itself off (goes to sleep) when not in use
- Low-battery indication
- Batteries easily changed – no need to open the case.

How it works

The circuit, built around a PIC16F690 microcontroller, is shown on the next page.

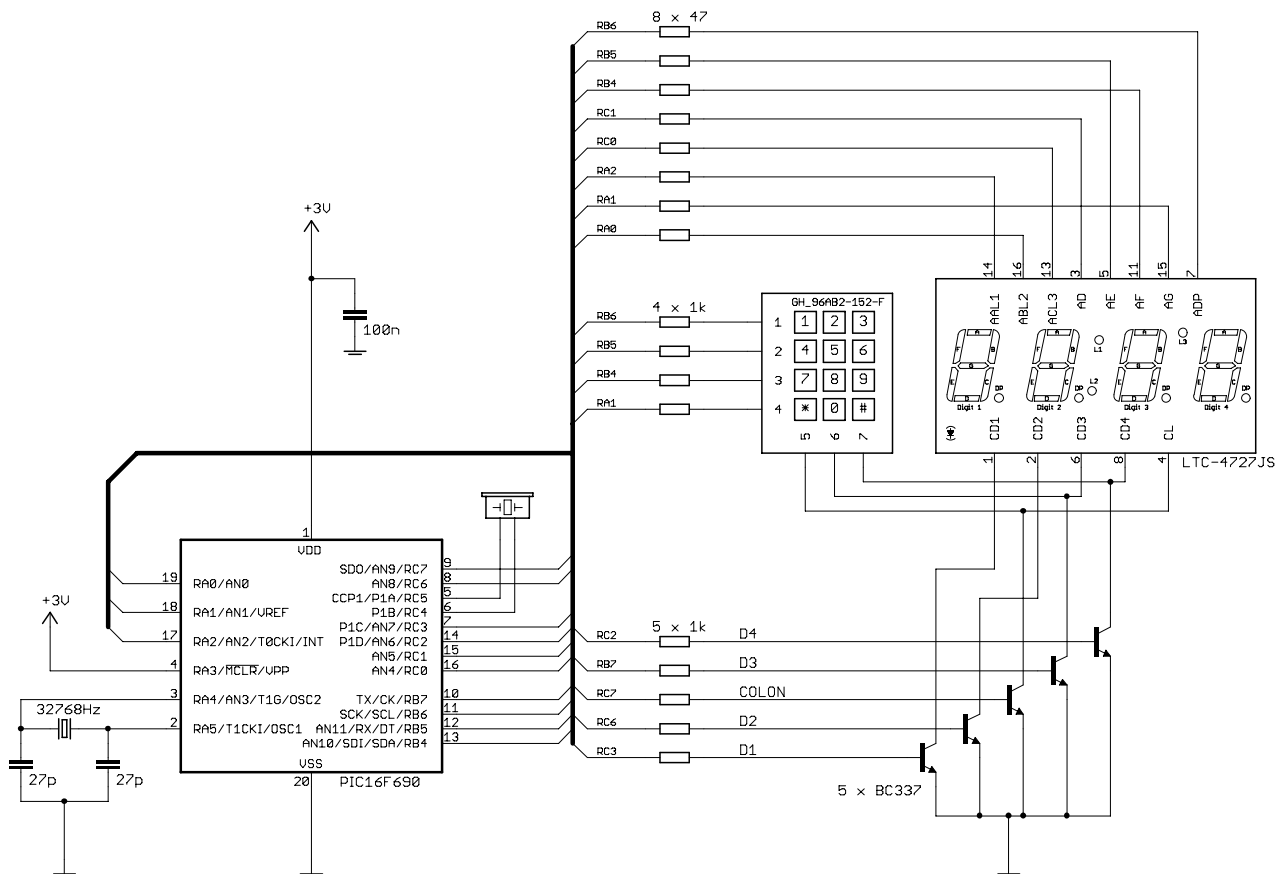
Two AA batteries provide a nominal 3V power supply. The PIC can operate down to 2.0V, so there is no need for regulation. However, as the battery voltage falls, the display will become dimmer. A “low battery” condition is indicated (by turning on the right-most decimal point) when the power supply falls to 2.4 V.

There is no power switch. Power is always supplied to the PIC, but it draws less than 10 μA when in standby mode, where the display is blanked and only the PIC's Timer1 oscillator, driven by the 32.768 kHz crystal (loaded by the two 27 pF capacitors), is running. The batteries should be able to supply this standby current for years.

A 100 nF bypass capacitor helps stabilise the power supply to the PIC.

The display module consists of four common-cathode 7-segment LED displays with the anodes (one per segment) connected in parallel. These anodes are driven directly by the 16F690, via 47 Ω resistors, which limit the current in each segment to 19 mA, assuming a 2.0 V forward voltage drop across each segment LED and an output high voltage on the PIC of 2.9 V. This is well within the PIC's maximum current ratings for each pin and in total.

NPN transistors, driven via 1 k Ω resistors, are used to connect each display cathode (one per digit, and another for the ':') to ground.



The displays are multiplexed; each transistor is turned on, one at a time, and the appropriate pattern output to the anode pins, to turn on the LED segments corresponding to a single digit to be displayed.

A timer-driven interrupt repeats this process almost 1000 times per second, meaning that, although only one digit is ever displayed at once, the overall display (all four digits plus ':') is refreshed almost 200 times per second; quickly enough that each digit appears to be continuously on.

The keypad consists of a four row \times three column matrix of pushbutton switches.

Three of the NPN transistors are used to pull the keypad columns low. Making the display-driving transistors do double-duty like this saves on transistors and PIC pins.

The keypad rows are connected to PIC pins, configured as digital inputs, with internal pull-ups enabled. This means that each pin will normally read high (or '1') until one of the keypad buttons, in whichever column is being pulled low, is pressed. By pulling each column low in turn, and reading the row inputs each time ('scanning' the keypad), it is possible to determine which keys are being pressed.

This scanning process is performed by the same interrupt routine which updates the display. Every 1.024 ms, it turns on one transistor, which pulls one display cathode and, for three of the transistors, one keypad column low. It then reads the keypad inputs (if applicable; not all of the transistors are connected to the keypad) and outputs the appropriate pattern for each digit to the display anodes. It also sounds the alarm, if it is enabled.

The alarm is a piezo speaker, driven by two PIC pins configured as complementary (one is the inverse of the other) PWM outputs, running at 4 kHz. Using complementary outputs in this way doubles the voltage seen by the piezo, significantly increasing the volume.

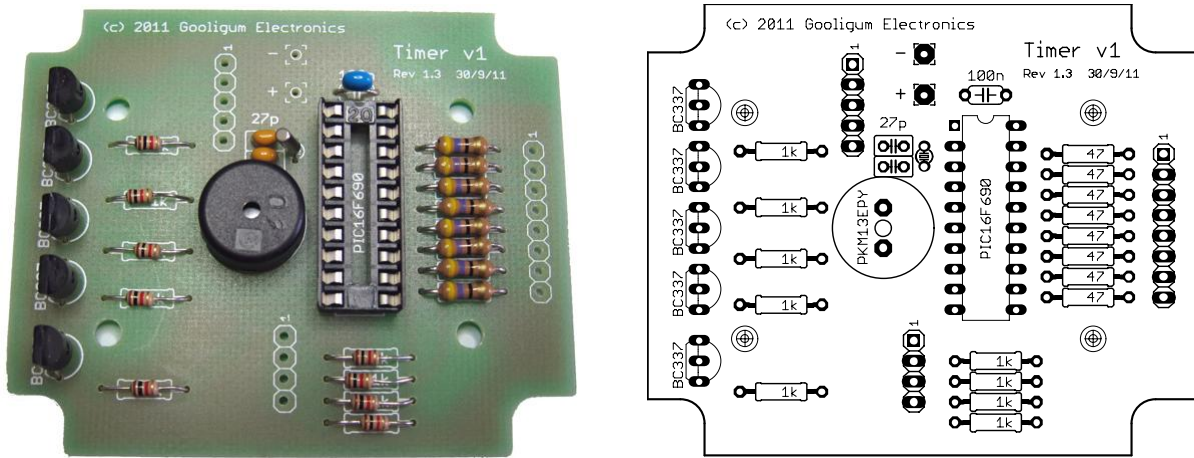
The PIC's Timer1 oscillator is used to drive the 32.768 kHz crystal, triggering an interrupt every two seconds, which in turn keeps track of the time.

As mentioned above, the Timer1 oscillator continues to run when the device is in standby (or sleep) mode. The PIC is configured to wake from sleep every two seconds, when TMR1 overflows. If the '*' key is pressed, the PIC stays awake and the minute timer comes back to life; otherwise it goes back to sleep for another two seconds.

Construction

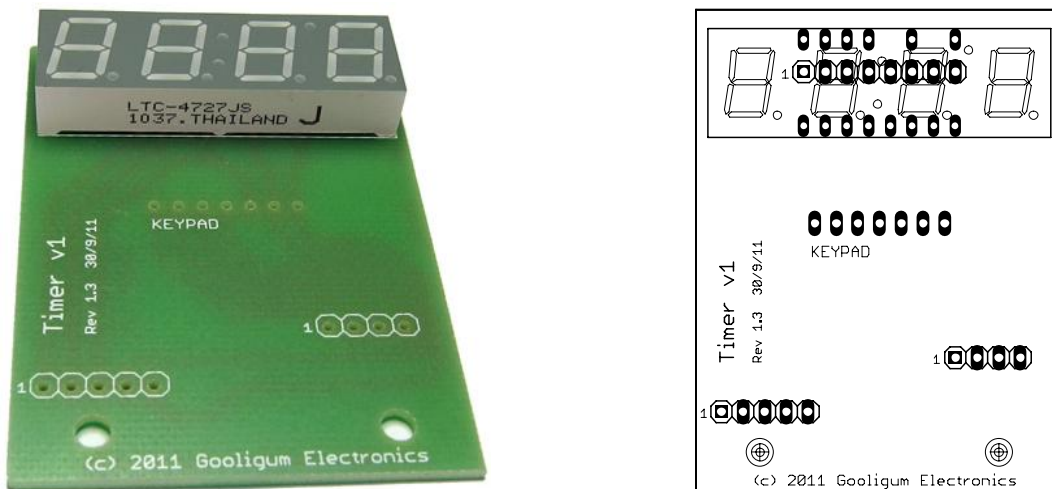
The kit contains two single-sided printed circuit boards, connected by ribbon cables – a main board holding the microcontroller, and a display board on which the display and keypad are mounted. If they are delivered as a single board, you will need to snap along the scoring to separate them.

The main board is shown below, with a component overlay diagram on the right.



It is often easiest to build up a PCB by starting with the lowest-profile components, so begin by soldering the resistors to the main board, followed by the capacitors and IC socket, and then the crystal and transistors – orienting them as shown in the overlay. Finally, install the buzzer. Note that the buzzer sits approximately 1 mm off the board.

The display board, with the display module fitted, and its component overlay, are shown below.



You can solder the 4-digit display module to the display board at this time, but before going further you need to prepare the enclosure.

The kit is designed to fit into a Hammond 1593Q hand-held instrument case, or equivalent. These cases include an inlay in the top section, a removable end panel (useful when they are used to build a remote control) and a battery compartment with a sliding door.

Print the drilling guide (downloadable from www.gooligum.com.au) and use a water-soluble glue to paste it to the inlay, with the LED display toward the removable panel end, as shown.



Use the guide to drill 4 x 2.5mm holes for the keypad mounting posts and to cut holes for the 4-digit LED display and keypad.

The cut-outs can be made fairly easily by drilling a series of small holes around the inside of the lines, cutting between the holes with a craft knife.

The slot for the keypad can be left rough, as it won't be visible.

Use a small file to slowly expand the display cut-out, regularly checking it against the display board until the display just fits, with the PCB lined up with the two mounting holes in the enclosure. Note that the other two mounting posts were within the display cut-out area and will have been removed.

Wash off the paper drilling guide with soapy water.

Temporarily press the keypad into place – it should be a tight fit – to check that the keypad pins correctly align with the holes in the display PCB and that the keypad pins reach far enough through the PCB to be soldered. They should just reach the copper layer. This is enough to make a good connection to each pad if you let the solder flow long enough. But to make certain, you may wish to lengthen the keypad pins by soldering solid wires (e.g. resistor lead cut-offs) to each.

Put a drop of glue (any suitable for plastics is ok, but “super glue” works well) on each keypad mounting hole, and press the keypad firmly into place.

You should also put a little of the same glue around the inside of the display cut-out, to glue the LED module to the case. A smear on each inside edge is plenty, if you've done a neat job of filing the cut-out to size.

Install the display PCB, with the display protruding a little past the outside of the case – just far enough to make it possible to solder the keypad pins. You can now screw the display PCB to the mounting holes, as shown.



The keypad pins can now be soldered to the display PCB.

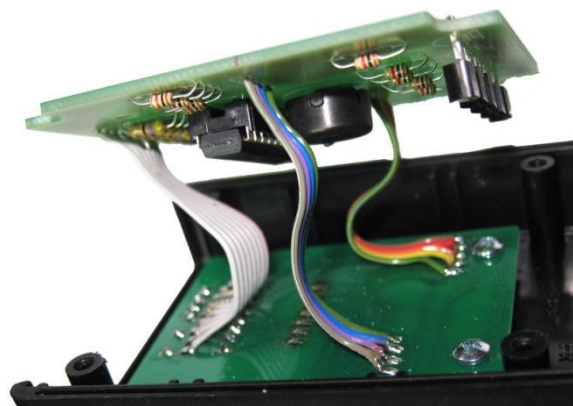
Prepare the three lengths of flat ribbon cable (8-way, 5-way, and 4-way), each around 5 cm long. Making them longer can make assembly a little easier, but if they are too long it will be difficult to fit them into the enclosure – there’s not a lot of room in there!

Solder the ribbon cables to the pads on the display board, as shown.



It is easiest to install the (programmed) PIC16F690 in its socket now, before the ribbon cables are connected to the main board.

You can now solder the ribbon cables to the main board, connecting the two PCBs together. For each connection, pin 1 (square) on the display board connects to pin 1 on the main board. The cables remain straight, not twisted, as shown.



Next, affix the battery holder to the case.

You can use some strong plastics glue to affix the holder to the battery compartment in the bottom half of the case, as shown here – but it has to be a very strong, specialised plastics glue, which “welds” the surfaces together, because it has to withstand the force of batteries being inserted and removed.

Unlike the keypad and display module, “super glue” is NOT good enough here!



A more robust solution is to use double-sided adhesive tape to attach the battery holder to the top half of the case, above the battery compartment.

To do so, you must cut off the keypad mounting posts (side cutters do a good job), and thread the battery holder leads past the enclosure mounting pillars, as shown.



Cut the battery holder leads to around 9 cm long and solder them to the pins marked '+' and '-' on the main PCB.

It is now possible to test the timer – see the operating instructions, below.

If the timer works correctly, you can go ahead and screw the main board to the bottom of the enclosure, using the four PCB mounting posts. You will need to swivel the top part of the enclosure out of the way as shown below – hopefully you've made the cables long enough!

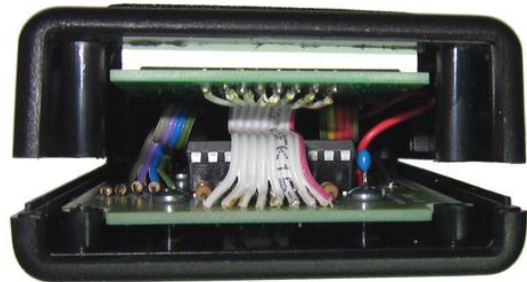


Tuck up the cables and then screw the enclosure halves together.

Remember to install the end panel first!

You can now test the timer again.

If the alarm is not loud enough, you can make it louder by drilling some holes in the end panel – after removing it from the case of course!



The full kit includes the top panel artwork, printed on an adhesive label. Alternatively, you can print the artwork from the PDF file available for download from www.gooligum.com.au.

Cut out the front label, removing the cut-outs for the display and keyboard with a sharp hobby knife. To make it easier to fit, you should cut around the outside of the lines.

You can now affix the label to the inlay section of the top panel, as shown.

Your timer is now ready to use!



Operation

Turn the minute timer on by pressing and holding the '*' key.

Use the numeric keypad to enter the time. The two digits to the right of the ':' represent hours and the two on the left are minutes. The maximum time you can enter is 99 hours and 99 minutes.

For example, if you want the alarm to sound after ninety minutes, type '90'. Or equivalently, you could type '130', which will show as "1:30", or one hour and thirty minutes.

To start the countdown, press '#'. The ':' on the display will flash, to show that the timer is running.

To pause the timer, press '#' again. The ':' will stop flashing and the time will stop counting down.

To resume the countdown, simply press '#' again.

When the alarm sounds, press '*' or '#' to silence it. Or wait twenty seconds and it will stop by itself.

Pressing '*' at any time will stop the timer and clear the display to "0:00". This is useful if you want to abort the countdown, or if you make a mistake when entering the time.

If the right-most decimal point lights, the batteries are getting low and it's time to change them!

To turn the timer off (place it in standby mode), press and hold the '*' key.

It will turn itself off, if no key is pressed for five minutes while the timer is stopped on "0:00".

Parts List

1	Pre-programmed PIC16F690-I/P
1	32768 Hz "watch" crystal (12.5 pF load capacitance)
2	27 pF ceramic capacitors
1	100 nF monolithic ceramic capacitor
8	47 Ω 1/4W resistors
9	1 k Ω 1/4W resistors
5	BC337 NPN transistors
1	10 mm 4-digit common cathode 7-segment LED display module (Lite-On LTC-4727JS)
1	3 \times 4 front-mount keypad (Grayhill 96AB2-152-F)
1	13 mm PCB ext drive piezo sounder, 5 mm spacing (e.g. muRata PKM13EPYH4000)
1	20-pin DIP IC socket
150 mm	8-way ribbon cable (or any suitable to make up 50 mm lengths of 8-way, 5-way and 4-way)
1	2-way AA flat battery holder with fly leads
1	Hand-held instrument case with battery compartment, at least 112 mm \times 66 mm \times 28 mm (e.g. Hammond 1593Q)
6	#4 or M2.5, 6 mm panhead machine screws
	Double-sided foam tape (optional, to mount battery holder – see text)

```

/*****
*
*   Filename:      Timer1
*   Date:         6/12/11
*   File Version: 1.6
*
*   Author:       David Meiklejohn
*   Company:      Gooligum Electronics
*
*****/
*
*   Architecture: Midrange PIC
*   Processor:    16F690
*   Compiler:     HI-TECH C v9.83 (Lite mode)
*
*****/
*
*   Files required: none
*
*****/
*
*   Description:   Simple minute timer
*
*   Sounds alarm when input time (hh:mm) counts down to 0:00
*
*   Timing is derived from 32.768 kHz crystal driving Timer1
*
*   Time is entered using 0-9 keys on 3x4 numeric keypad,
*   with each keystroke displayed as a digit on the right,
*   pushing existing digits to the left.
*
*   Timer is started and stopped using '#' key
*
*   Display ':' flashes at 1 Hz when timer is running,
*   steady when timer not running.
*
*   Alarm sounds when timer counts down to "0:00"
*   Alarm stops sounding when '*' or '#' is pressed,
*   or after time-out period of 20 seconds
*
*   '*' resets time to "0:00" and stops timer
*   and turns off alarm (if it is sounding)
*
*   Holding '*' down for 2 secs (at any time),
*   or not pressing any key for 5 mins (while display is zeroed)
*   places timer into sleep mode
*
*   Pressing '*' again (after release) wakes the device
*
*   Low battery condition (Vdd < 2.4 V)
*   indicated by decimal point on digit 4
*
*****/
*
*   Pin assignments:
*
*       RB4-6, RA1      = keypad row inputs (active low)
*       RC2,7, RB7     = keypad column enables (active high)

```

```
*      RA0-2, RB4-6, RC0,1 = 7-segment display bus (active high)      *
*      RB7, RC2,3,6,7      = display segment enables (active high)  *
*      P1A, P1B            = piezo speaker                          *
*      OSC1, OSC2          = 32.768 kHz crystal                      *
*                                                                    *
*****/
```

```
#include <htc.h>
```

```
/****** CONFIGURATION *****/
```

```
#ifdef __DEBUG
```

```
// no code or data protect, no brownout detect, no watchdog, no power-up timer
// int reset, int clock with I/O, failsafe clock monitor, int/ext switch disabled
```

```
__CONFIG(CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF & PWRTE_OFF &
         MCLR_OFF & FOSC_INTRCIO & FCMEN_ON & IESO_OFF);
```

```
#else
```

```
// code and data protect, no brownout detect, no watchdog, power-up timer
// int reset, int clock with I/O, failsafe clock monitor, int/ext switch disabled
```

```
__CONFIG(CP_ON & CPD_ON & BOREN_OFF & WDTE_OFF & PWRTE_ON &
         MCLR_OFF & FOSC_INTRCIO & FCMEN_ON & IESO_OFF);
```

```
#endif
```

```
// *** Pin assignments
```

```
// keypad row inputs (active low):
```

```
#define KP_R1      RB6          // row 1 (1,2,3), pin 1
#define KP_R2      RB5          // row 2 (4,5,6), pin 2
#define KP_R3      RB4          // row 3 (7,8,9), pin 3
#define KP_R4      RA1          // row 4 (*,0,#), pin 4
```

```
// keypad column enables (active high):
```

```
#define KP_C1      RC7          // column 1 (1,4,7,*), pin 5
#define sKP_C1     sPORTC.RC7
#define KP_C2      RB7          // column 2 (2,5,8,0), pin 6
#define sKP_C2     sPORTB.RB7
#define KP_C3      RC2          // column 3 (3,6,9,#), pin 7
#define sKP_C3     sPORTC.RC2
```

```
// 7-segment display digit enables (active high):
```

```
#define D1_EN      RC3          // 1: 10 hours
#define sD1_EN     sPORTC.RC3
#define D2_EN      RC6          // 2: 1 hours
#define sD2_EN     sPORTC.RC6
#define D3_EN      RB7          // 3: 10 mins
#define sD3_EN     sPORTB.RB7
#define D4_EN      RC2          // 4: 1 mins
#define sD4_EN     sPORTC.RC2
#define CL_EN      RC7          // 5: colon and indicator
#define sCL_EN     sPORTC.RC7
```

```
// 7-segment display indicators (active high):
```

```
#define DP         RB6          // decimal point (on each digit)
#define sDP        sPORTB.RB6
#define L1         RA2          // colon top
#define sL1        sPORTA.RA2
#define L2         RA0          // colon bottom
```

```

#define sL2          sPORTA.RA0
#define L3           RC0          // indicator LED
#define sL3          sPORTC.RC0

// 7-segment display bus TRIS masks (for enabling/disabling display output / keypad row pins)
#define t7SEG_A      0b000111    // RA0-2
#define t7SEG_B      0b01110000  // RB4-6
#define t7SEG_C      0b00000011  // RC0-1

/***** SYMBOLS *****/

// key codes (returned by keypad scan function)
#define KEY_STAR     10          // '*' key
#define KEY_HASH     11          // '#' key
#define KEY_NONE     12          // no key pressed

// display codes (sent to digit display function)
#define DSP_STAR     10          // '*'
#define DSP_HASH     11          // '#'
#define DSP_BLANK    12          // blank digit

/***** CONSTANTS *****/
#define MAX_DB_CNT   50/5        // key debounce period = 50 ms / 5 ms per sample
#define MAX_STAR_CNT 2000/5      // time to press '*' key to enter standby
                                // = 2 sec ( / 5 ms per count)
#define ALM_TIMEOUT  20          // seconds to sound alarm
#define KEY_TIMEOUT  300         // seconds to enter standby if no key pressed
#define MINVDD       2400        // minimum Vdd in mV (for low battery test)

/***** FUNCTION PROTOTYPES *****/
void initialise();              // initialise processor
void set7seg(char digit);      // display digit on 7-segment display

/***** GLOBAL VARIABLES *****/

// variables updated by ISR
volatile unsigned char db_key = KEY_NONE; // most recent debounced keypress:
                                           // 0-9 = '0' to '9'
                                           // KEY_STAR = '*',
                                           // KEY_HASH = '#',
                                           // KEY_NONE = no key pressed

volatile bit          key_change = 0;      // keypad state change flag:
                                           // 1 =
                                           // changed

volatile unsigned int press_cnt = 0;       // measures keypress time
                                           // (increments every 5 ms while any key is
                                           // held down,
                                           // cleared on release)

volatile unsigned char digit[4] = {0,0,0,0}; // display digits
volatile unsigned char hrs = 0;           // time remaining
volatile unsigned char mins = 0;
volatile unsigned char secs = 0;

```

```
volatile unsigned int    t_secs = KEY_TIMEOUT;    // timeout counter (seconds)

// flags
volatile bit            buzz_on = 0;              // buzzer state (1 = on)
volatile bit            timer_on = 0;            // timer state (1 = countdown active)
volatile bit            standby = 0;            // sleep mode (1 = in standby mode)
volatile bit            low_bat = 0;            // low battery detected (1)

// shadow registers
volatile union {                                     // PORTA
    unsigned char        byte;
    struct {
        unsigned         RA0      : 1;
        unsigned         RA1      : 1;
        unsigned         RA2      : 1;
        unsigned         RA3      : 1;
        unsigned         RA4      : 1;
        unsigned         RA5      : 1;
    };
} sPORTA;

volatile union {                                     // PORTB
    unsigned char        byte;
    struct {
        unsigned         : 4;
        unsigned         RB4      : 1;
        unsigned         RB5      : 1;
        unsigned         RB6      : 1;
        unsigned         RB7      : 1;
    };
} sPORTB;

volatile union {                                     // PORTC
    unsigned char        byte;
    struct {
        unsigned         RC0      : 1;
        unsigned         RC1      : 1;
        unsigned         RC2      : 1;
        unsigned         RC3      : 1;
        unsigned         RC4      : 1;
        unsigned         RC5      : 1;
        unsigned         RC6      : 1;
        unsigned         RC7      : 1;
    };
} sPORTC;

/***** MAIN PROGRAM *****/
void main()
{
    unsigned char        key;                       // current keypress

    initialise();                                   // configure ports and
    timers
```

```

// enable interrupts
PEIE = 1; // enable peripheral
ei(); // and global interrupts

// Main loop
for (;;)
{
    // handle any key presses (detected by TMR0 interrupt)
    if (key_change)
    {
        key = db_key; // get new keystroke
        key_change = 0; // clear change flag
        t_secs = KEY_TIMEOUT; // restart keypad timeout counter
        switch (key) // and handle it:
        {
            case KEY_NONE: // key release:
                break; // do nothing

            case KEY_HASH: // '#' key:
                if (buzz_on)
                    buzz_on = 0; // turn off buzzer if sounding
                else
                    timer_on = !timer_on; // else toggle timer on/off
                break;

            case KEY_STAR: // '*' key:
                timer_on = 0; // disable and
                digit[0] = 0; // zero timer
                digit[1] = 0;
                digit[2] = 0;
                digit[3] = 0;
                hrs = 0;
                mins = 0;
                buzz_on = 0; // turn off buzzer
                break;

            default: // other keys (0-9):
                digit[0] = digit[1]; // show new digit on display
                digit[1] = digit[2]; // shift digits to the left
                digit[2] = digit[3];
                digit[3] = key; // display new key press on right
                hrs = digit[0]*10 + digit[1]; // update time from new digits
                mins = digit[2]*10 + digit[3];
                secs = 60; // (new time is shown a full minute
                // before it decrements)
                break;
        }
    }
}

// update display digits with current time
// NOTE: time is maintained by TMR1 interrupt
// digits are displayed by TMR0 interrupt
digit[0] = (unsigned)hrs / 10;
digit[1] = (unsigned)hrs % 10;
digit[2] = (unsigned)mins / 10;
digit[3] = (unsigned)mins % 10;

```



```

// check for shutdown ('*' held down or keypad timeout)
if ((db_key == KEY_STAR && press_cnt > MAX_STAR_CNT) || t_secs == 0)
{
    // blank display until key release
    buzz_on = 0; // turn off alarm
    standby = 1; // flag standby mode (will blank display)
    digit[0] = 0; // zero timer
    digit[1] = 0;
    digit[2] = 0;
    digit[3] = 0;
    hrs = 0;
    mins = 0;
    while (db_key != KEY_NONE) // wait for key release
        ;

    // prepare to enter standby mode
    TMR2ON = 0; // disable PWM
    TMR1 = 0; // reset Timer1 and interrupt flag,
    TMR1IF = 0; // so that Timer1 wakeup will be 2 secs
    from now
    TOIE = 0; // ensure no wake-ups from Timer0
    di(); // disable interrupts

    // sleep until '*' is pressed again
    do
    {
        // minimise power
        PORTA = 0; // drive all pins low
        TRISA = 0;
        PORTB = 0;
        TRISB = 0;
        PORTC = 0;
        TRISC = 0;

        // enter standby mode (for 2 secs)
        SLEEP(); // sleep until woken by Timer1
        TMR1IF = 0; // clear interrupt flag for next time

        // setup to read '*' key
        TRISA = t7SEG_A; // make display bus / keypad row pins inputs
        TRISB = t7SEG_B;
        TRISC = t7SEG_C;
        KP_C1 = 1; // enable keypad column 1 ('*' is C1, R4)
    }
    while (KP_R4); // sleep again if '*' not pressed
    db_key = KEY_STAR; // record keypress

    // restart display and keyboard scanning
    initialise(); // reset processor state
    ei(); // re-enable interrupts
    standby = 0; // clearing standby mode will re-enable
    display

    // wait for '*' key release
    while (db_key != KEY_NONE)
        ;
}

```

```

        t_secs = KEY_TIMEOUT;           // restart keypad timeout counter
    }
}

/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt_isr(void)
{
    static unsigned char    mpx_cnt = 0;           // multiplex counter
    static unsigned char    db_cnt = MAX_DB_CNT;  // debounce counter
    static unsigned char    key;                 // current keypress

    if (T0IF)
    {
        // *** Service Timer0 interrupt
        //
        // This interrupt is triggered by a TMR0 overflow every 1.024 ms
        //
        // Displays current time on 4-digit 7-segment LED display
        //     Implements multiplexing; on successive interrupts:
        //         digit[0-3] output to digit 1-4
        //         ':' turned on, or flashed when timer enabled
        //     Entire display refresh every 5 interrupts (5.12 ms)
        //     Output changes are made via shadow registers
        //
        // Scans keypad
        //     Each column read in turn, during display multiplexing "loop"
        //     Scanning complete after 5 interrupts (5.12 ms)
        //     Debounce performed at end of scan
        //     -> updates keypad variables:
        //         db_key      (current debounced keypad state)
        //         key_change  (new keypress or release indication)
        //         press_cnt   (time current key has been held down)
        //
        // Sounds alarm
        //     Enables PWM (piezo) outputs, every 0.5 s, if alarm is on
        //
        // Displays indicator if battery is low
        //
        // Blanks display on shutdown (standby)
        //
        // Copies shadow registers to output ports
        //
        //
        T0IF = 0;                       // clear interrupt flag

        // Setup for keypad read
        //     blank display during keypad read
        //     (re-enable display output at end of ISR)
        TRISA |= t7SEG_A;                // make display bus / keypad row pins inputs
        TRISB |= t7SEG_B;
        TRISC |= t7SEG_C;

        // Multiplexing "loop" used to:
        //     display digits on 4 x 7-segment LED display
        //     handle ':' flashing
    }
}

```

```
// scan and debounce keypad
// handle alarm
//
// mpx_cnt determines current keypad column to read and/or digit to display
//

switch (mpx_cnt)
{
    case 0:
        // setup for keypad read cycle
        key = KEY_NONE;

        // read keypad column 1
        // (pulled low for ':' display during last interrupt)
        if (!KP_R1) key = 1;
        if (!KP_R2) key = 4;
        if (!KP_R3) key = 7;
        if (!KP_R4) key = KEY_STAR;

        // display digit 1
        if (digit[0] > 0) // output digit 1
            set7seg(digit[0]); // (blank if leading zero)
        else
            set7seg(DSP_BLANK);
        sD1_EN = 1; // enable digit 1

        // sound alarm (if enabled)
        if (buzz_on && TMR1 & 1<<14) // sound alarm each second if alarm flag
            set
            { // (TMR1<14> cycles at 1 Hz)
                TRISC4 = 0; // enable PWM (piezo) outputs to
                sound alarm
                TRISC5 = 0;
            }
        else
        {
            TRISC4 = 1; // disable PWM outputs to silence alarm
            TRISC5 = 1;
        }

        break;

    case 1:
        // display digit 2
        set7seg(digit[1]); // output digit 2
        sD2_EN = 1; // enable digit 2

        break;

    case 2:
        // display digit 3
        set7seg(digit[2]); // output digit 3
        sD3_EN = 1; // enable digit 3

        break;

    case 3:
```

```

// read keypad column 2
// (pulled low for digit 3 display during last interrupt)
if (!KP_R1) key = 2;
if (!KP_R2) key = 5;
if (!KP_R3) key = 8;
if (!KP_R4) key = 0;

// display digit 4
set7seg(digit[3]); // output digit 4
sD4_EN = 1; // enable digit 4

// low battery indication
if (low_bat) // if low battery,
    sDP = 1; // turn on decimal point on digit 4

break;

case 4:
// read keypad column 3
// (pulled low for digit 4 display during last interrupt)
if (!KP_R1) key = 3;
if (!KP_R2) key = 6;
if (!KP_R3) key = 9;
if (!KP_R4) key = KEY_HASH;

// scan is complete, so debounce keypad
key_change = 0;
if (key == db_key) // if no change
    db_cnt = MAX_DB_CNT; // reset debounce count
else // if change
    if (--db_cnt == 0) // when change is stable
    {
        db_key = key; // accept it
        key_change = 1; // flag it
        db_cnt = MAX_DB_CNT; // reset debounce count
    }

// update key press counter
if (key_change) // if a new key is pressed or released
    press_cnt = 0; // reset the count
else
    if (db_key != KEY_NONE) // if a key is currently pressed
        press_cnt++; // increment the count

// display colon
sPORTA.byte = 0; // clear all display bits
sPORTB.byte = 0;
sPORTC.byte = 0;
sL1 = 1; // turn on colon LEDs
sL2 = 1;
if (timer_on && TMR1 & 1<<14) // flash colon if timer enabled
{ // (TMR1<14> cycles at 1
    Hz)
    sL1 = 0; // blank colon LEDs every alternate
    second
    sL2 = 0;
}

```

```
sCL_EN = 1; // enable colon

break;
}

// increment multiplex counter, to select next time slice for next interrupt
mpx_cnt++;
if (mpx_cnt == 5) // reset count if at end of sequence
    mpx_cnt = 0;

// blank display if entering standby mode (provides visual indication)
if (standby)
{
    sPORTA.byte &= ~(t7SEG_A); // clear display bus port bits
    sPORTB.byte &= ~(t7SEG_B);
    sPORTC.byte &= ~(t7SEG_C);
}

// copy shadow regs to ports
PORTA = sPORTA.byte;
PORTB = sPORTB.byte;
PORTC = sPORTC.byte;

// start driving 7-segment display bus (display will light)
TRISA &= ~(t7SEG_A); // make display bus pins outputs
TRISB &= ~(t7SEG_B);
TRISC &= ~(t7SEG_C);
}

if (TMR1IF)
{
    // *** Service Timer1 interrupt
    //
    // This interrupt is triggered by a TMR1 overflow every 2 s
    //
    // Handles background time keeping:
    //     If timer active:
    //         Countdown time
    //         Stop timer and enable alarm when countdown complete
    //     If timer not active and time is 0:00
    //         Countdown keypad timeout
    //     If alarm sounding:
    //         Countdown alarm timeout
    //         Disable alarm when timeout complete
    //     If in standby mode:
    //         Do nothing
    //
    // Initiates low battery check (ADC read of 0.6 V ref)
    //
    //
    TMR1IF = 0; // clear interrupt flag

    // If preparing to enter standby mode, do nothing
    if (standby)
        return;

    // If timer active, countdown time
```

```
if (timer_on)
{
    if (hrs > 0 || mins > 0)
    {
        // decrement time by 2 secs
        if (secs > 0)
            secs -= 2;
        else // handle underflow
        {
            secs = 58;
            if (mins == 0)
            {
                mins = 59;
                if (hrs > 0)
                    hrs--;
            }
            else
                mins--;
        }
    }
    // check for countdown complete
    if (hrs == 0 && mins == 0)
    {
        buzz_on = 1; // turn on alarm
        secs = ALM_TIMEOUT; // set seconds to countdown alarm timeout
        timer_on = 0; // turn off timer
    }
}
// If timer not active and time is 0:00, countdown keypad timeout
else
    if (hrs == 0 && mins == 0 && t_secs > 0)
        t_secs -= 2;

// If alarm sounding, countdown alarm timeout
if (buzz_on)
{
    // decrement alarm timeout by 2 secs
    if (secs > 0)
        secs -= 2;
    else
        // timeout complete
        buzz_on = 0; // turn off alarm
}

// Initiate battery check
GO = 1; // initiate ADC conversion (will trigger interrupt)
}

if (ADIF)
{
    // *** Service ADC interrupt
    //
    // Triggered every 2 secs by Timer1 interrupt
    //
    // Compares sampled value of 0.6 V reference with a threshold,
    // to test for low battery (Vdd) condition

```

```

//
ADIF = 0; // clear interrupt flag

// test for low Vdd (measured 0.6 V > threshold)
low_bat = (ADRESH > 255*600L/MINVDD);
}
}

/***** FUNCTIONS *****/

/***** Initialise processor *****/
void initialise()
{
// configure ports
PORTA = 0; // start with all digits off
PORTB = 0; // -> all display pins low
PORTC = 0;
sPORTA.byte = 0; // update shadow registers
sPORTB.byte = 0;
sPORTC.byte = 0;
TRISA = 0b111000; // output pins are RA0-2,
TRISB = 0b00001111; // RB4-7,
TRISC = 0b00110000; // RC0-3, 6-7
ANSEL = 0; // all inputs are digital
ANSELH = 0;
WPUA = 0b00000010; // enable weak pull-ups on RA1
WPUB = 0b01110000; // and RB4-6

// setup timers + weak pull-ups
OPTION_REG = 0b01000001; // configure Timer0 and weak pull-ups:
//0----- enable global weak pull-ups (*RABPU = 0)
//--0----- timer mode (T0CS = 0)
//----0--- prescaler assigned to Timer0 (PSA = 0)
//-----001 prescale = 4 (PS = 001)
// -> increment every 4 us
// -> TMR0 overflows every 1.024 ms
TOIE = 1; // enable Timer0 interrupt

T1CON = 0b00001110; // configure Timer1:
//-0----- gate disabled (TMR1GE = 0)
//--00---- prescale = 1 (T1CKPS = 00)
//----1--- LP oscillator enabled (T1OSCEN = 1)
//-----1-- asynchronous mode (/T1SYNC = 1)
//-----1- external clock (TMR1CS = 1)
//-----0 disable Timer1 (TMR1ON = 0)
// -> increment TMR1 at 32.768 kHz
TMR1 = 0; // clear Timer1 (start counting from 0)
TMR1ON = 1; // start Timer1
TMR1IE = 1; // enable Timer1 interrupt

// setup PWM
T2CONbits.T2CKPS = 0; // TMR2 prescale = 1
// -> TMR2 increments every 1 us
PR2 = 249; // period PR2+1 = 250 us
// -> PWM frequency = 4 kHz

```

```

CCPR1L = 125; // CCPR1L:CCP1CON<5:4> = 500
CCP1CONbits.DC1B = 0; // -> PWM pulse width = 125 us
// -> PWM duty cycle = 50%
CCP1CONbits.CCP1M = 0b1101; // select PWM mode:
// P1A active-high, P1B active-low
CCP1CONbits.P1M = 0b00; // single output
STRA = 1; // enable P1A and P1B as PWM outputs
STRB = 1; // (output pins disabled by TRIS bits)
TMR2ON = 1; // start PWM (enable TMR2)

// setup ADC (for low battery test)
ADCON1 = 0b01010000; // configure ADC:
// -101---- A/D conversion clock = Fosc/16
// -> Tad = 4.0 us (with Fosc = 4 MHz)
ADCON0 = 0b00110101;
// 0----- MSB of result in ADRESH<7> (ADFM = 0)
// -0----- voltage reference is Vdd (VCFG = 0)
// --1101-- select 0.6 V reference (CHS = 1101)
// -----1 turn ADC on (ADON = 1)
ADIF = 0; // clear ADC interrupt flag
ADIE = 1; // enable ADC interrupt
}

```

```

/***** Display digit on 7-segment display *****/

```

```

void set7seg(char digit)

```

```

{

```

```

// Lookup pattern table for 7 segment display on PORTA

```

```

// RA2:0 = AGB

```

```

const char pat7segA[13] = {
//   AGB
0b000101, // 0
0b000001, // 1
0b000111, // 2
0b000111, // 3
0b000011, // 4
0b000110, // 5
0b000110, // 6
0b000101, // 7
0b000111, // 8
0b000111, // 9
0b000011, // *
0b000110, // #
0b000000 // (none)
};

```

```

// Lookup pattern table for 7 segment display on PORTB

```

```

// RB5:4 = EF

```

```

const char pat7segB[13] = {
//   EF
0b00110000, // 0
0b00000000, // 1
0b00100000, // 2
0b00000000, // 3
0b00010000, // 4
0b00010000, // 5
0b00110000, // 6
};

```



```
    0b00000000,    // 7
    0b00110000,    // 8
    0b00010000,    // 9
    0b00110000,    // *
    0b00000000,    // #
    0b00000000    // (none)
};

// Lookup pattern table for 7 segment display on PORTC
// RC1:0 = DC
const char pat7segC[13] = {
    //      DC
    0b00000011,    // 0
    0b00000001,    // 1
    0b00000010,    // 2
    0b00000011,    // 3
    0b00000001,    // 4
    0b00000011,    // 5
    0b00000011,    // 6
    0b00000001,    // 7
    0b00000011,    // 8
    0b00000011,    // 9
    0b00000001,    // *
    0b00000010,    // #
    0b00000000    // (none)
};

// lookup pattern bits and write to shadow registers
sPORTA.byte = pat7segA[digit];
sPORTB.byte = pat7segB[digit];
sPORTC.byte = pat7segC[digit];
}
```