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 date 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 2or 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 – 0Fh	General Purpose Registers	27h – 2Fh	Map to Bank 0 07h – 0Fh
10h – 1Fh	General Purpose Registers	30h – 3Fh	General Purpose Registers

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 GPIO[1].

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	$\overline{\text{TO}}$	$\overline{\text{PD}}$	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 10^{th} 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`' – "**ret**urn with **l**iteral 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 × 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.

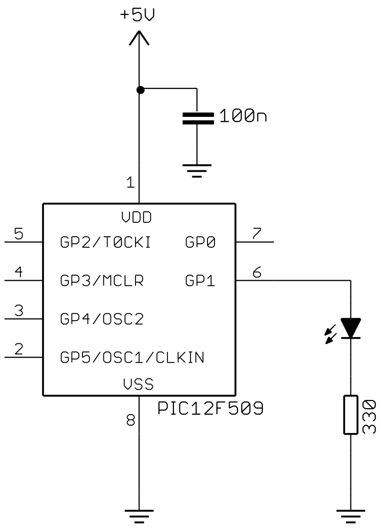
To count multiples of 10 ms, we need to add a third loop counter, as in the following code:

```
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 page[4]. 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 section[5]. 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 × 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

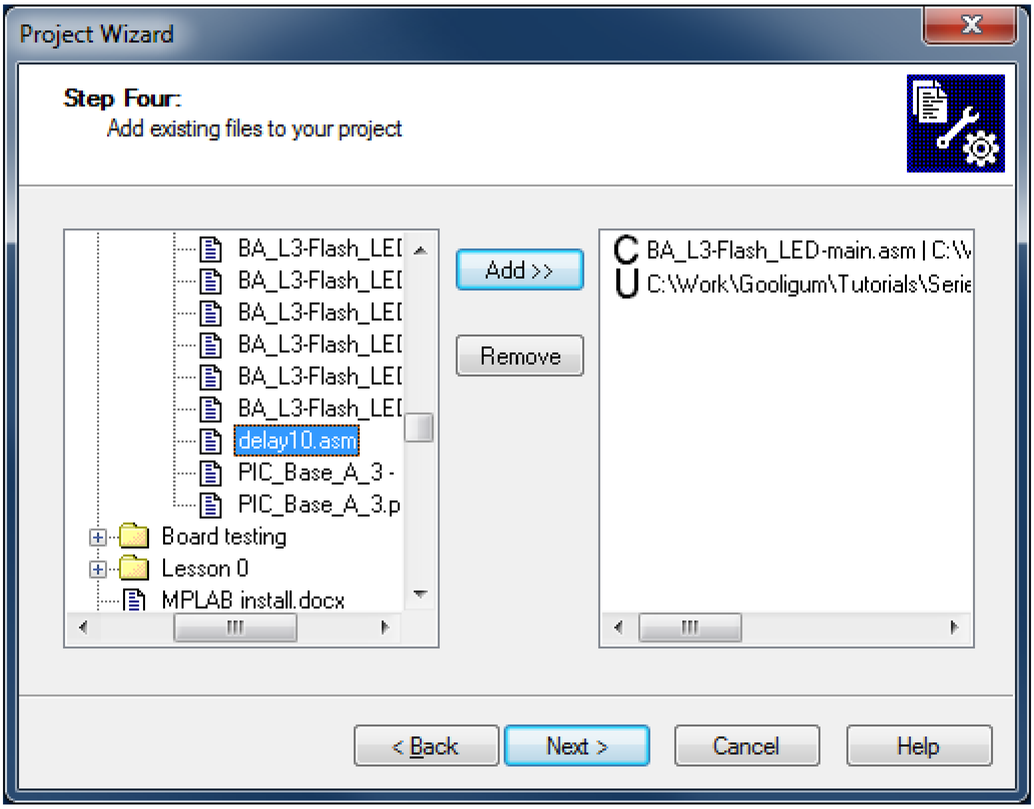
To create the multiple-file project, open an existing project and then save it with a new name, such as "BA_L3-Flash_LED-mod", in the same way as you did when creating a new project in lesson 2.

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".

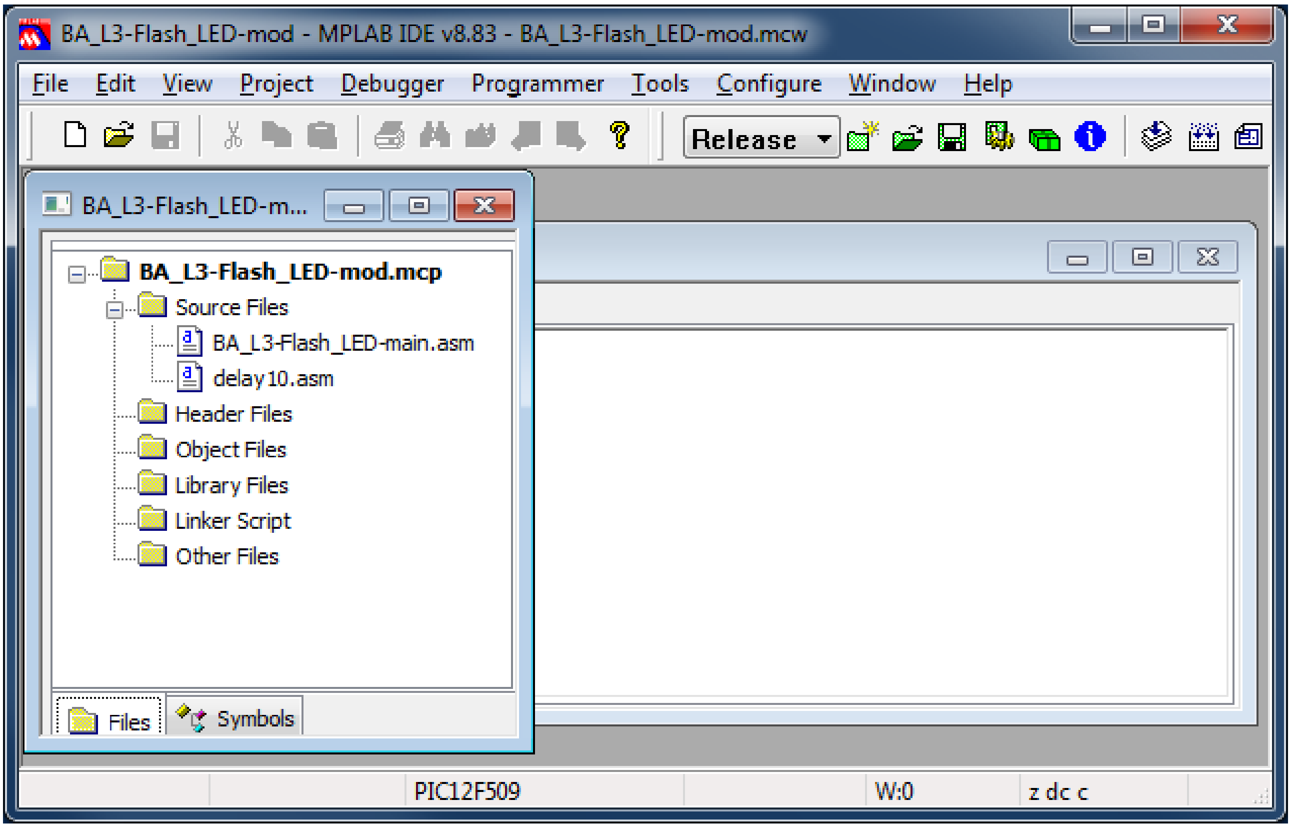
Next close the editor window and run the project wizard to reconfigure the active project, as before.

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:

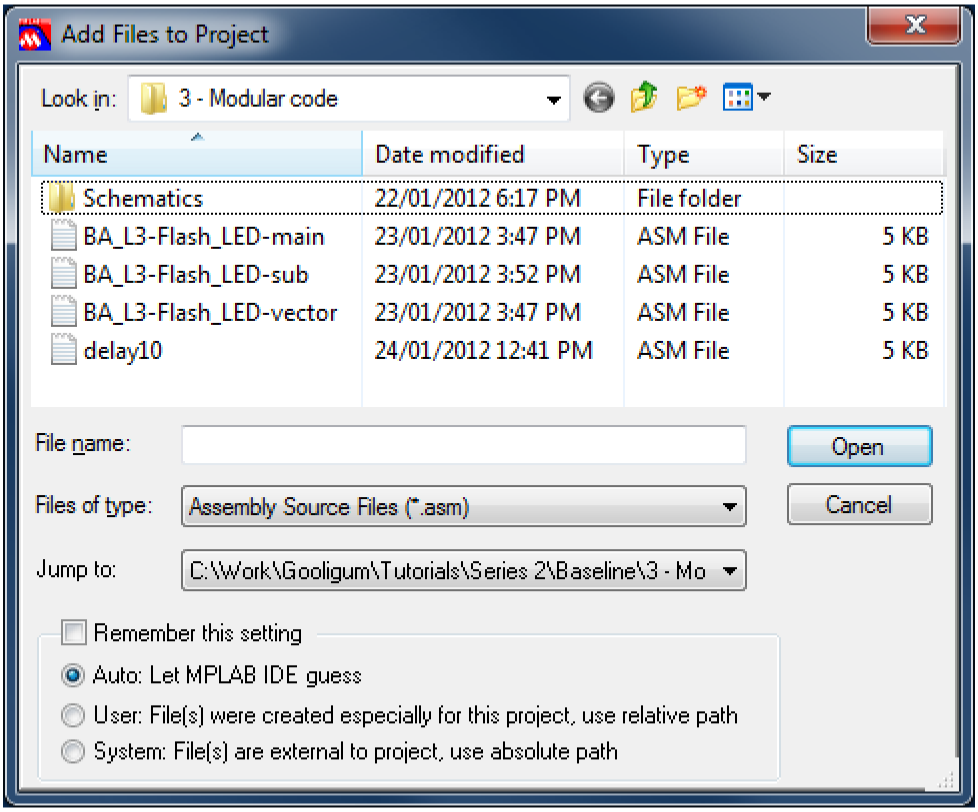


After clicking "Next >" and then "Finish", you will see that your project now contains both source files:



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.

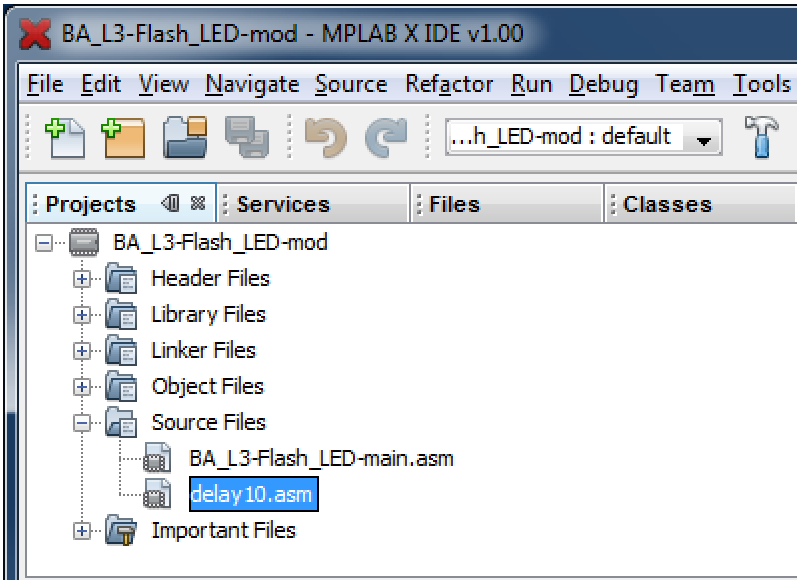
There are a few ways to do this, but the easiest is probably to right-click the source file in the Projects window and select "Copy".

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".

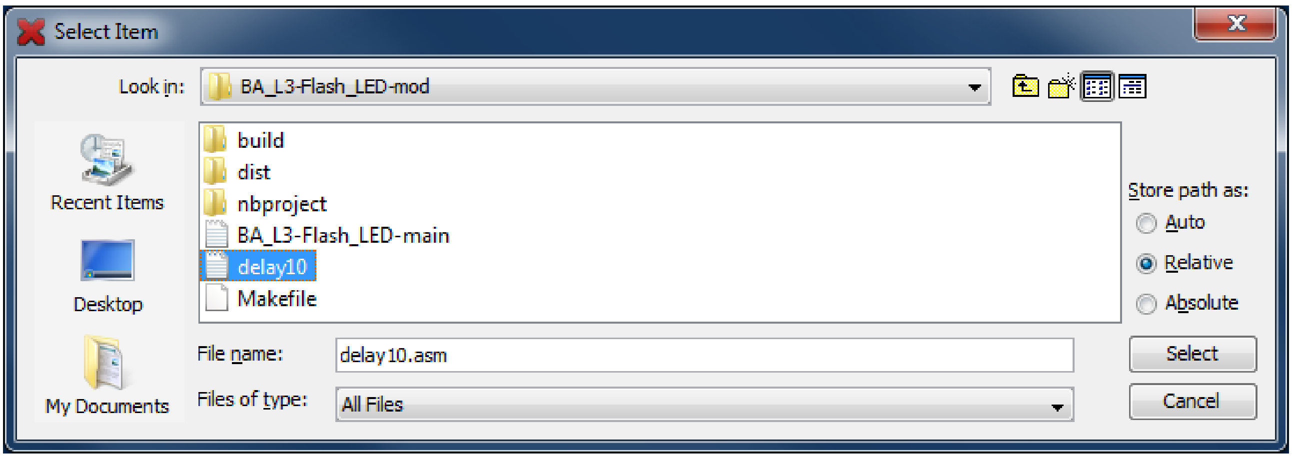
Your project should look like the one shown on the right.



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".

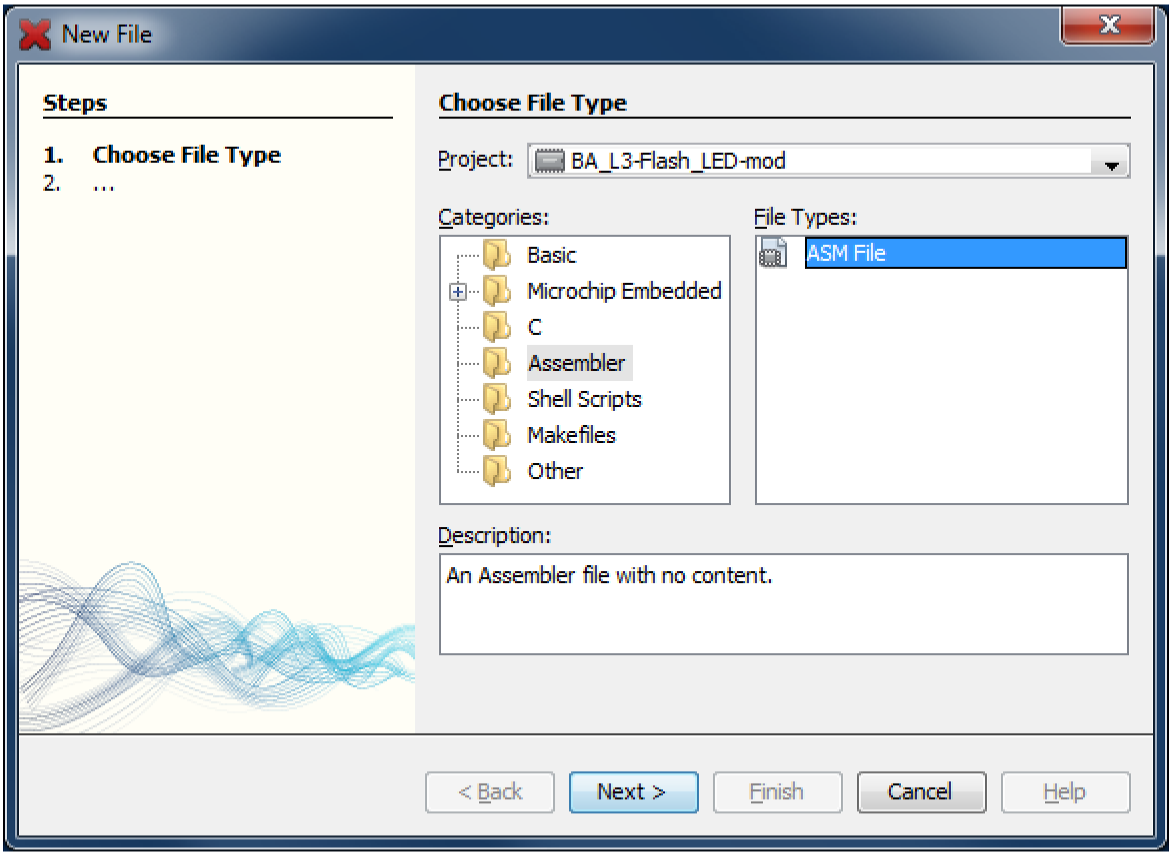
The only problem is that this new source file hasn't appeared in the Projects window; MPLAB X doesn't yet know that the new file is part of the project. So, we need to add it.

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:

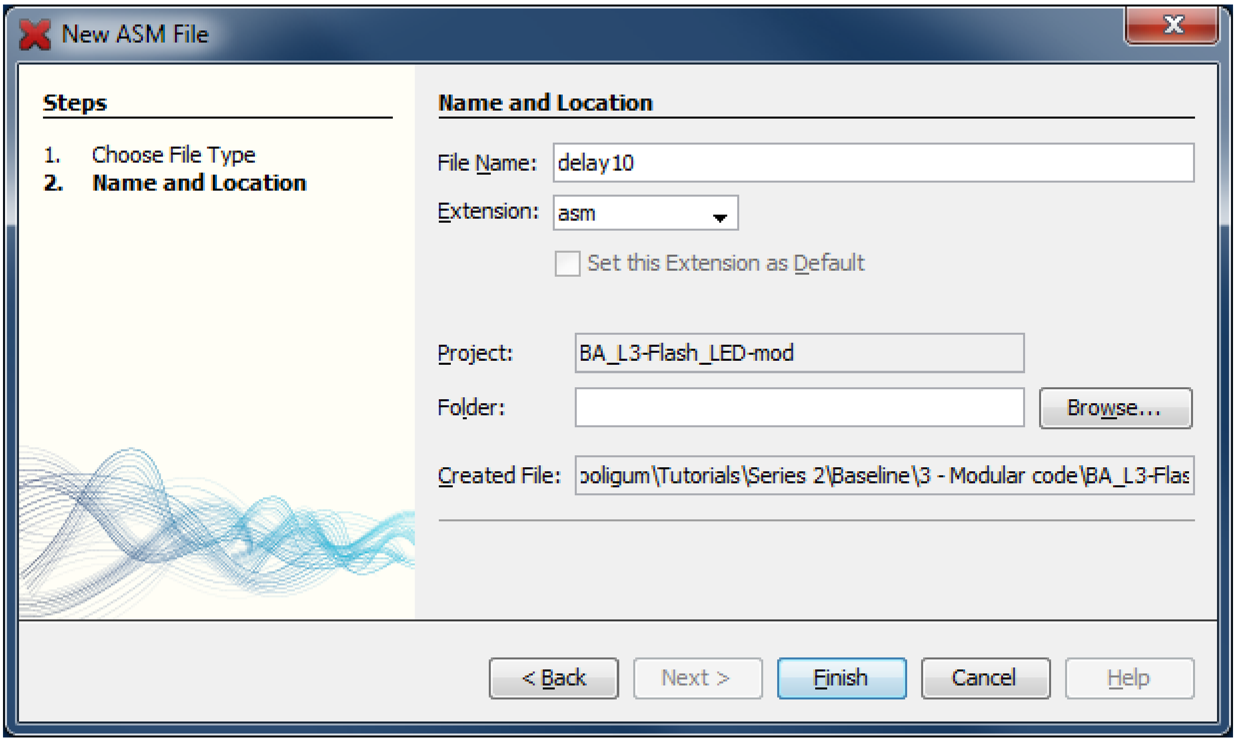


As you can see, it gives you the option to specify whether the file has a relative path (appropriate for most "user" files) or absolute path (for most "system" files). If you're unsure, just select "Auto" and let MPLAB decide.

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.

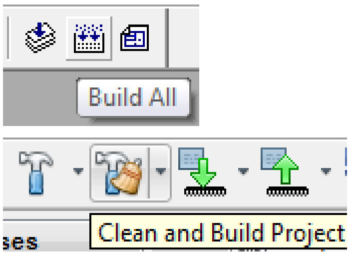
Note that a subroutine vector is still used (to avoid potential problems due to the baseline architecture's subroutine addressing limitation, explained earlier), as it is not possible to know where in program memory the linker will place the module.

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:

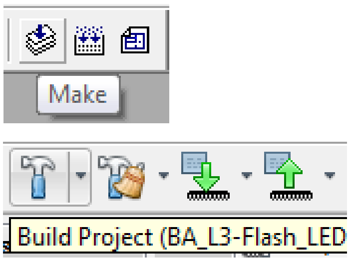
- The `GLOBAL` and `EXTERN` directives work as a pair.
- `GLOBAL` is used in the file that defines a module, to export a symbol for use by other modules.
- `EXTERN` is used when calling external modules. It declares that a symbol has been defined elsewhere.

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 directory[6]. 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[7].

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 'mplink' 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.