

Introduction to PIC Programming

Midrange 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.

As we saw in [baseline lesson 3](#), 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 midrange 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 midrange PICs; there is no reason to use jump tables, as did for the baseline devices.

The midrange PICs have eight stack registers, compared with only two in the baseline architecture. This means 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.

Midrange 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 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

    return

```

Example 1: Flashing an LED (20% duty cycle)

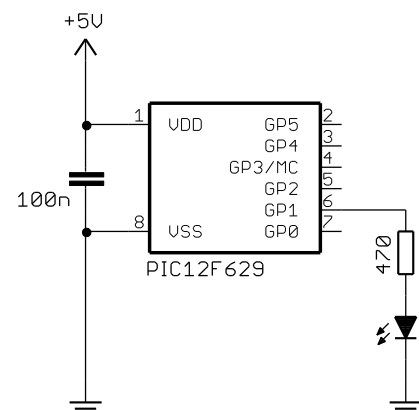
In [lesson 1](#), we used the fixed 500 ms delay routine to flash the LED in this simple circuit (below) at 1 Hz, with a 50% duty cycle.

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 baseline 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                ; 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

    return

```

This routine can then be called, from the main loop, to generate the 200 ms and 800 ms delays we need, as follows:

```

    banksel  GPIO      ; select bank for GPIO access
flash  movlw    1<<GP1  ; turn on LED on GP1
    movwf    GPIO
    movlw    .20       ; stay on for 0.2s:
    call    delay10    ; delay 20 x 10ms = 200ms
    clrf    GPIO      ; turn off LED
    movlw    .80       ; stay off for 0.8s:
    call    delay10    ; delay 80 x 10ms = 800ms

    goto    flash      ; repeat forever

```

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 before 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, 4Mhz 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        CODE    0x0000      ; processor reset vector
                ; calibrate internal RC oscillator
                call    0x03FF      ; retrieve factory calibration value
                banksel OSCCAL      ; then update OSCCAL
                movwf   OSCCAL

                ; initialisation
                movlw   ~(1<<GP1)   ; configure GP1 (only) as an output
                banksel TRISIO
                movwf   TRISIO

;***** Main loop
flash        banksel GPIO        ; select bank for GPIO access

                movlw   1<<GP1      ; turn on LED
                movwf   GPIO
                movlw   .20         ; stay on for 0.2s:
                call    delay10     ; delay 20 x 10ms = 200ms
                clrf    GPIO        ; turn off LED
                movlw   .80         ; stay off for 0.8s:
                call    delay10     ; delay 80 x 10ms = 800ms

                goto    flash       ; repeat forever

```

```

;***** Subroutines
delay10      movwf    dc3      ; delay W x 10ms
              ; 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

              clr     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 the midrange architecture, 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 midrange 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 midrange 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 midrange 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 covered 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 midrange 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. So, 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 you should always use it, regardless of which midrange 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 midrange 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 midrange 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 midrange 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

;*****
          CODE
delay10   ; delay W x 10ms
          movwf    dc3          ; -> ?+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

          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 midrange 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 midrange PIC; it's not specific to any, so you can use the include file for any of the midrange 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 [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 on "Source Files" in the project window, and then selecting "Add Files" 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:         13/7/08
;  File Version:  1.0
;
;  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 10ms 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, 4Mhz 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   CODE    0x0000      ; processor reset vector
        ; calibrate internal RC oscillator
        call    0x03FF      ; retrieve factory calibration value
        banksel OSCCAL      ; then update OSCCAL
        movwf   OSCCAL

        ; initialisation
        movlw   ~(1<<GP1)   ; configure GP1 (only) as an output
        banksel TRISIO
        movwf   TRISIO

        clrf    sGPIO       ; start with shadow GPIO zeroed

;***** Main loop
flash   ; toggle LED
        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

        ; delay 500 ms -> 1 Hz at 50% duty cycle
        movlw   .50
        pagesel delay10
        call    delay10

        ; repeat forever
        pagesel flash
        goto    flash

        END

```

We've been flashing LEDs for a while; it's time to move on to something else.

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.