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 compiler (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 MCLR input enabled, even though no connection to the MCLR (GP3) pin is shown in the circuit diagram above. That’s because the 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 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:

```c
void main()
{
    // 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:

```c
{  
    // 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:

```assembly
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:

```assembly
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:

```assembly
;*****************************************
;                             *
;   Description:    Variable Delay : N x 1 seconds (1 - 255 secs)    *
;   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
```
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
EXTERN      delay1s_R               ; (calls delay1s)

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 & _WDT
```

As in the C example above, note that we’ve selected external reset, with the MCLR input enabled, even though no connection to the MCLR (GP3) pin is shown in the circuit diagram. What’s not shown is that, on a development board, the 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           ; effect
    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:

```assembly
;***** 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:
    ; --0------ timer mode (T0CS = 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 kΩ 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 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 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:

```pseudo
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
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:

```pseudo
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
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:

```c
#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:

```c
#define DEBOUNCE 10*1000/256    // switch debounce count = 10 ms/(256us/tick)
#define DbnceHi(PIN) TMR0 = 0;                /* re
                    // set timer              */
                    /* wait until debounce time */
                    /*   if input low,          */
                    /*     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:

```c
#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 operation:

```c
#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:

```c
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-up 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     -> 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)
DbnceHi(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) 
            { 
                LEDs = 0;               // turn off all LEDs
                G_LED = 1;              // turn on green LED
            }
            if (sec_cnt == Y_START) 
            { 
                LEDs = 0;               // turn off all LEDs
```

```
Y_LED = 1;              // turn on yellow LED
}
if (sec_cnt == R_START)
{
    LEDS = 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
        LEDS = 0;            // turn off all LEDs
        DbnceHi(BUTTON);    // wait for stable button release
        SLEEP();            // enter sleep mode
    }
}
}

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
;
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
              goto    start
    goto    start
    movf    TMR0,w ; has switch 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:

```assembly
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:

```assembly
;***** 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:

```assembly
#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:

```assembly
#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:

```assembly
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:

```assembly
constant POLL_MS = 50               ; polling interval (in ms)
```

We also need to define the variables that we will be using:

```assembly
;***** 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)
DbnceHi 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    LEDS            ; 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    LEDS            ; 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    LEDS            ; 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:

```assembly
;*** 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):

```assembly
;***** 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:

```assembly
;*** 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:

```assembly
;*** 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 necessarily need to make that decision up front, leaving the implementation details for later.

So we can define the timeout value simply as:

```plaintext
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:

```plaintext
// configure hardware

// ensure that pushbutton is not pressed

// initialise time count

time_cnt = 0
```

Main loop:

```plaintext
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
  DbnceHi(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()`:

```c
**** 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:

```c
// 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)
        { LEDS = 0;               // turn off all LEDs
          G_LED = 1;              // turn on green LED
          }
    if (sec_cnt == Y_START)
        { LEDS = 0;               // turn off all LEDs
          Y_LED = 1;              // turn on yellow LED
          }
    if (sec_cnt == R_START)
        { LEDS = 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:

```c
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:

```assembly
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:

```assembly
; 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:

```assembly
;*** 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:

```assembly
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 PICs. 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”:

```c
// 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:

```c
//*** Main loop
for (;;) {
  // wait for button press
  while (BUTTON)              // wait until button low
    ;

  // then turn off whichever LED is currently lit:
  LEDS = 0;                   // turn off all LEDs

  // 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;
  }

  // we then wait for stable button release, and debounce it:
  // wait for stable button release
  DbnceHi(BUTTON);
```

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:

```c
// 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:

```assembler
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”:

```plaintext
; 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:

```plaintext
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:

```plaintext
clrf    LEDS            ; 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:

```plaintext
; 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:

```plaintext
;*** 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:

```c
// 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:

```c
#include <xc.h>
#include <stdint.h>
#include "stdmacros-SC8.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)
    // 0-0------ enable weak pull-ups (/GPPU = 0)
    // 0-0------ timer mode (T0CS = 0)
    // 0-0------ prescaler assigned to Timer0 (PSA = 0)
    // 0-0------ -> 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;
                    break;

                    case YELLOW:         // if yellow:
                        state = RED;       // next state = red
                        R_LED = 1;          // turn on red LED
                        break;
                    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)
{
    LEDS = 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:

```assembly
list        p=10F200
#include    <p10F200.inc>```
errorlevel -312 ; no "page or bank selection not needed" messages

#include <stdmacros-base.inc> ; DbncHi - 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
go to start ; jump to main code

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

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

;***** Initialisation
start
; configure port
```assembly
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 LEDS ; 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
```
```asm
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
    decf  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 12F508\(^5\). 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:

```c
// 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
  end

  // delay 1 second while polling switches
  repeat 1000/N times
    delay N ms
  end

  // check for button press
  if button pressed
    enter standby mode
  end

  // check for mode change
  if SELECT = manual
    time_cnt = 0  // reset timeout counter
    exit  // exit automatic mode
  end

  // 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:

```c
***** 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:

```c
#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:

```c
//**** PROTOTYPES *****/
void AutoMode(void);                // automatic mode
void ManualMode(void);              // manual mode
void standby(void);                 // enter standby (low-power) mode
```

The main loop is simply:

```c
//*** 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:

```c
/***** 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’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:

    ```c
    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’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 (`btfss` and `btfsc`) 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:

```plaintext
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), 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:

```plaintext
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:

```assembly
;***** 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:

```assembly
;***** 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’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 LEDS
    bsf G_LED ; turn on green LED (only)
```
Again most of the loop code is taken directly from the program we developed in step 5, so we’ll only list the comments for most of the unchanged sections here, except for the extra code within the polling loop which reads the mode switch and exits the subroutine if automatic mode has been selected:

```
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
; 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
;*** Repeat (until mode change or timeout)
goto man_start
```

**Final Step: Production version**

We now have a working set of traffic lights which meets all of our design requirements – but it’s still only a prototype. As a final step, we need to make the design production-ready; something that can be manufactured (or built as a kit, in this case) at reasonably low cost while being reliable enough.

We’ll need a power supply. We’ve said that the traffic lights should be battery powered, so we’ll use batteries, but what type and how many? A 4 × AAA battery holder was selected for the Gooligum traffic lights kit because it is a suitable size – square, and fits nicely at the base of the enclosure. It would have been possible to use fewer batteries, but the weight of four AAA batteries adds stability. These decisions aren’t always based on electrical considerations!

However, 4 × AAA batteries supply a nominal 6.0 V (a little more for fresh alkaline cells), while the PIC12F508 has a maximum VDD of 5.5 V (the absolute maximum specified in the data sheet is 6.5 V). To reduce the power supply to below 5.5 V, and also provide reverse polarity protection (to avoid destroying the PIC if the batteries are inserted backwards, a rectifier diode is added in line with the battery supply, as shown in the circuit diagram below:
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 Ω to increase the LED current; with a power supply of 5.3 V (6.0 V supplied by the batteries minus a drop of 0.7 V across the diode) and an LED forward voltage of 1.8 V, the diode current will be (5.3 V – 1.8 V) / 180 Ω = 19.4 mA. That’s well within the 25 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:

```c
// 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:

```c
#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:

```c
// configure ports
GPIO = 0b111111;            // start with all LEDs off
TRIS = 0b001110;            // configure LED pins (GP0,4,5) as outputs
```

And then throughout the rest of the code, to turn off all the LEDs we use:

```c
LEDS = 0b111111;            // turn off all LEDs
```

Note that we could instead use:

```c
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:

```c
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 = 0b111111; // start with all LEDs off
    TRIS = 0b001110; // 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)
        // 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)
```c
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) {
            LEDs = 0b111111; // turn off all LEDs
            G_LED = 0; // turn on green LED
        }
        if (sec_cnt == Y_START) {
            LEDs = 0b111; // turn off all LEDs
            Y_LED = 0; // turn on yellow LED
        }
        if (sec_cnt == R_START) {
            LEDs = 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
    LEDs = 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
            LEDS = 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
            DbncHi(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)
{
    LEDS = 0b111111; // turn off all LEDs
    DbncHi(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:

```assembly
; pin assignments
#define LEDS      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:

```assembly
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:

```assembly
; 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:

```assembly
movlw   b'111111'       ; turn off all LEDs
movwf   LEDS
```

Since we only have LEDs on GP0, GP4 and GP5, we could instead use:

```assembly
movlw   b'110001'       ; turn off all LEDs
movwf   LEDS
```

But as noted for the C version, it would be harder to maintain.

Finally to turn on a single LED we have, for example:

```assembly
bcf     G_LED           ; turn on green LED
```

This means that, to turn off all the LEDs except one, we have for example:

```assembly
movlw   b'111111'       ; turn off all LEDs
movwf   LEDS
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:

```assembly
; pin assignments
#define LEDS      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)
```
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:

```assembly
;************************************************************************
; 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 LEDS 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 ; 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

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

;***** Initialisation

start

; 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

; 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'111111' ; 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 ; sec_cnt = 0
clrf sec_cnt
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
Introductory 8-bit PIC Project 1: Traffic Lights

**** Manual mode

ManualMode_R

; Initialise variables
banksel time_cnt
clr time_cnt ; time_cnt = 0
clr 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

clr time_cnt ; reset timeout counter
clr 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:
oxorf state,w
btfs 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:
oxorf state,w
btfs 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…

Introductory 8-bit PIC Project 1: Traffic Lights Page 56