Flashing LEDs

We’re going to experiment with flashing some LEDs in this part. First, we will just flash two alternate LEDs, then make a simple set of traffic lights, and finally a running pattern; you all remember Knight Rider don’t you?

To keep things simple, circuit diagrams will no longer show the ICSP connections as it just makes the diagrams harder to read. If you’ve kept your circuit on the breadboard from part #2, you can re-use that.

Alternate LED flashing

There are a number of ways of doing this, and we will examine two of them.

Fig1. Circuit 3a – Flash two LEDs

The circuit from part #2 had a single flashing LED. To make this second LED flash, at the same rate as the first LED is rather simpler; we just add additional HIGH and LOW instructions like this:

```
-- Configure I/O ports and pins here
Output PORTC.4  ' Tell the compiler that PORTC.4 is an output
Output PORTC.5  ' Likewise, make PORTC.4 an output

'-- Start of main program
FlashAgain:
  High PORTC.4  ' Set PORTC.4 = Logic HIGH = LED On - LED #1
  High PORTC.5  ' Set PORTC.3 = Logic HIGH = LED On - LED #2
```
DelayMS 250 ' Delay for 250ms (1/4 second)
Low PORTC.4 ' Set PORTC.4 = Logic LOW = LED Off - LED #1
Low PORTC.5 ' Set PORTC.5 = Logic LOW = LED Off - LED #2
DelayMS 250 ' Delay for 250ms (1/4 second)
GoTo FlashAgain ' Jump back to the start of the routine, and start again

You can see that three additional instructions have been added to control LED #2; highlighted in yellow. We first tell the compiler that PORTC bit 5 will now be used as an output, and then proceed to set it’s logic level HIGH and LOW as required.

We could of course swap them around so that when LED#1 is on, LED#2 is off and via versa. A good idea on some PIC projects is to have a heart-beat LED. This is a LED that toggles on and off every so often so that you know everything is working correctly.

Another way of writing this program would be:

'-- Configure I/O ports and pins here
Output PORTC.4 ' Tell the compiler that PORTC.4 is an output
Output PORTC.5 ' Likewise, make PORTC.4 an output
High PORTC.4 ' Set PORTC.4 = Logic HIGH = LED On - LED #1
High PORTC.5 ' Set PORTC.5 = Logic HIGH = LED On - LED #2

'-- Start ot main program
FlashAgain:
DelayMS 250 ' Delay for 250ms (1/4 second)
Toggle PORTC.4 ' Whatever the logic level of PORTC.4 is, invert or toggle it.
Toggle PORTC.5 ' Whatever the logic level of PORTC.5 is, invert or toggle it.
GoTo FlashAgain ' Jump back to the start of the routine, and start again

Now, the first two highlighted lines force the PORTC.4 & 5 bits to logic HIGH – LED on. We then execute the 250ms delay, then a new instruction is executed TOGGLE. This tells the compiler to simply invert the current logic level of the I/O port.

Changing the program to look like the above saves 4 program words of storage which doesn’t seem like a lot, but every word counts, but also means we must be executing fewer instructions, so the program will run more efficiently.

One thing to be careful off is how the PIC initialises everything when it starts up. I/O port bits and other registers can be in a random state.

Because of the two HIGH instructions executed when the program starts, the LEDs will be forced on, then after a 250ms delay, the LEDs will both go off, then on, and repeat.

This is “deterministic” behaviour because we force the LEDs on initially, we always know what their status will be from this point on.

However, if we removed the two HIGH instructions, then when the PIC starts, one or both LEDs may be on initially because of the random way the PIC starts up. So instead of the LEDs flashing
on and off together in sequence, one LED could actually end up always being the opposite state to the other. For a heart-beat type monitor LED, this may not be of any concern and removing those two HIGH instructions actually saves another eight program words.

If the intention was to create alternating LEDs, the circuit could be changed to this:

--- Configure I/O ports and pins here
Output PORTC.4  ' Tell the compiler that PORTC.4 is an output
High PORTC.4    ' Set PORTC.4 = Logic HIGH = LED #1 On, LED #2 Off

--- Start of main program
FlashAgain:
DelayMS 250     ' Delay for 250ms (1/4 second)
Toggle PORTC.4  ' Whatever the logic level of PORTC.4 is, invert or toggle it.
    ' A LOW will turn off LED #1, but turn on LED #2
    ' A HIGH will turn on LED #1, but turn off LED #2
GoTo FlashAgain  ' Jump back to the start of the routine, and start again

This has the advantage that it saves one I/O pin (PORTC.5 is now free) and saves another 2 instruction words.
Traffic Lights.

The sequence of traffic lights is a simple pattern that can easily be replicated using a PIC and three coloured LEDs.

The sequence we will use starts at Red. After a delay the display changes to Red + Amber, a small delay and then to Green. After another longish delay, the display changes to just Amber with a small delay and then back to Red. The sequence then repeats.

We’ve covered all the commands required to accomplish this task and if you want to go off and have a go yourself at doing this, it’s a fairly simple exercise if you think about what needs to be done carefully.

An example of how the program could be written is on the next page.

But before you do, it’s worth mentioning program loops.

PIC programs don’t usually ever end; that is, they run around monitoring things, twiddling with I/O ports, waiting for switches to be pressed etc, but the program they run never ends.

It’s sometimes possible that whilst your experimenting you may want your PIC to complete a specific task like wait for a switch to be pressed, then light a LED and you’ve then finished, and there are two BASIC instructions to tell the CPU that you’ve done; STOP and END. However, if you read their description in the AMICUS18 manual, you will find that both instructions place the microcontroller in an infinite loop. The only way out of this loop is to reset the PIC (or cause an interrupt).

NEVER allow your programs to just run off the bottom of your code. By that I mean your programs should either have a loop that takes them back to someplace within the program, or at worst case, execute an END or STOP statement. If the PIC finds a way out of your program, it will start executing random program instructions formed from whatever junk is left in unused parts of memory, and this can cause havoc with your circuit.
PIC Projects – Part #3  Flashing LEDs.

'-- Configure I/O ports and pins here
   Output PORTC   ' Make ALL of PORTC an output

'-- Start of main program
   ' Make sure that all the LEDs are off as we don't know how the PIC will power up
   Low PORTC.0  ' Red
   Low PORTC.1  ' Amber
   Low PORTC.2  ' Green

DoAgain:
   High PORTC.0  ' Red On
   DelayMS 5000  ' Wait for 5 seconds
   High PORTC.1  ' Amber on (remember the RED is still on)
   DelayMS 1000  ' 1 second delay for Red + Amber
   Low PORTC.0  ' Red Off
   Low PORTC.1  ' Amber Off
   High PORTC.2  ' Green On
   DelayMS 5000  ' Wait for 5 seconds
   Low PORTC.2  ' Green Off
   High PORTC.1  ' Amber on (remember the RED is still on)
   DelayMS 1000  ' 1 second delay for Amber only
   Low PORTC.1  ' Amber Off

   ' The next part of the sequence is back to just Red on.
   ' So we jump back to the start of the loop where the Red is switched back on
   GoTo DoAgain

If you work through the sample code above, it should all be quite straightforward.

There is one last thing we could do to make this code a lot simpler to read and maintain.
Saying PORTC.2 or PORTB.7 is all well and good, but once you start using PORTB.5 and PORTA.2
it’s all going to get very confusing and difficult trying to remember what each port bit is being
used for.

BASIC allows for us to assign a meaningful name or label to things; we can “alias” them using
the SYMBOL directive. SYMBOL isn’t an instruction and it doesn’t generate any code so you can
use them as often as you like. Look at the re-written version of the above code using SYMBOLs.

'-- Configure I/O ports and pins here
   Output PORTC   ' Make ALL of PORTC an output
       SYMBOL LED_RED  PORTC.0
       SYMBOL LED_AMBER  PORTC.1
       SYMBOL LED_GREEN  PORTC.2

'-- Start of main program
   ' Make sure that all the LEDs are off as we don't know how the PIC will power up
   Low LED_RED
   Low LED_AMBER
   Low LED_GREEN

DoAgain:
   High LED_RED
   DelayMS 5000  ' Wait for 5 seconds
   High LED_AMBER  ' Amber on (remember the RED is still on)
   DelayMS 1000  ' 1 second delay for Red + Amber
   Low LED_RED
Low LED_AMBER
High LED_GREEN

DelayMS 5000  ' Wait for 5 seconds

Low LED_GREEN
High LED_AMBER  ' Amber on (remember the RED is still on)

DelayMS 1000  ' 1 second delay for Amber only

Low PORTC.1  ' Amber Off

' The next part of the sequence is back to just Red on.
' So we jump back to the start of the loop where the Red is switched back on
GoTo DoAgain

The three additional lines at the start of our program are the aliases we will be using. These lines simply tell the compiler that whenever the piece of text “LED_RED” is seen, to replace it with PORTC.0, and so on. Using SYMBOLs has no detrimental impact on your program in any way and they should always where ever possible as it makes reading and maintaining your program a lot simpler. Imagine that for some reason, the above program needs to be changed to use PORTB instead of PORTC. There are just three SYMBOL lines now that need to be updated.

If you’ve built this circuit on a bread-board then we will be modifying it slightly in part 4 where we look at push switches and inputs.
Knight Rider moving LED display.

Everybody remembers Knight Rider from the 1980’s and the moving red light display on the front of his black car, well, that sequence is really easy to do with a PIC and eight red LEDs.

Change the circuit to use eight red LEDs, connect from each pin on PORTC, via a 470R resistor to 0v.

`'-- Configure I/O ports and pins here
    Output PORTC     ' Make ALL of PORTC an output

Symbol RED_RED5  PORTC.4

'-- Variable Declaration
Dim Loop          As Byte

'-- Start of main program
DoAgain:
    For Loop = 0 To 7 Step 1  ' Count up from 0 to 7 in steps of 1
        Low RED_LEDS  ' Force all the PORTC outputs to LOW - All LEDs will now be off
        SetBit RED_LEDS, Loop  ' Set the bit, whose number is specified in Loop, HIGH
        DelayMS 50  ' We need to leave the LED lit for a while else our eyes won't see it
    Next Loop

    For Loop = 7 To 0 Step -1  ' Count down from 7 to 1
        Low RED_LEDS  ' Force all the PORTC outputs to LOW - All LEDs will now be off
        SetBit RED_LEDS, Loop  ' Set the bit, whose number is specified in Loop, HIGH
        DelayMS 50  ' We need to leave the LED lit for a while else our eyes won't see it
    Next Loop

    GoTo DoAgain

Ok, a couple of new instructions in this program need to be explained.

**DIM ... AS BYTE**

This means dimension, which isn’t very helpful really. What we are actually asking the compiler to do, is reserve a little chunk of RAM where we can store a piece of information; a variable.

We’ve assigned a label, “Loop” to this location in RAM, and we have no idea where in RAM the compiler will place this variable; and usually we don’t care.

Instead of Loop, we could have called it wibble, fred22, JIM_121_SILLY or any other name that meets the requirements of the compiler; I chose Loop as it make more sense than wibble.

The size of the chunk of RAM to be reserved is controled by the variable type; in this case a Byte (a byte is 8 bits long).

It’s called a variable because the data stored in the location assigned to us by the compiler can be variable; can be changed, to whatever we wish to store there, within the rules allowed by a byte that is. A byte can contain any positive number from 0 to 255.
FOR
This is a really useful BASIC instruction and there is much more information available in the AMICUS18 manual for this.

However, in brief this program line means start a counter, store the current value of the counter in RAM at a location we’ve previously specified called Loop. The counter in this case will start from (be initialised), at 0, and finish at 7.

FOR will always increment by +1 unless you specify otherwise. This means that in reality, the Step +1 is not required and can be removed. The compiler is quite clever and will remove this automatically, so it won’t use any more program space if you feel more comfortable leaving it in.

SETBIT
This isn’t an instruction you would usually find in BASIC. This tells the compiler to set a specified bit in a variable to 1. SETBIT always sets a bit to 1. If you want to set a bit to 0 you use the instruction CLEARBIT.
Remember that a bit can ONLY be a 0 or a 1. There is no other possible value.

The program line:

```plaintext
SetBit PORTC, Loop ' Set the bit, whose number is specified in Loop, HIGH
```

is telling the compiler to set a bit in PORTC (and the bit number required is specified in the variable Loop). Remember that the FOR instruction previously was set to count from 0 to 7. PORTC is also 8 bits wide and bit positions always start from 0.

NEXT
This is an end of loop instruction and tells the compiler that when it reaches this point, it should obey the STEP value supplied in the related FOR instruction, and then jump back to the line AFTER the FOR instruction.

So when the compiler sees the NEXT instruction, it will look at the value at location Loop and increment it by 1. If the value in Loop is now less than, or equal to the maximum value allowed (in this case 7 because that’s what we specified), then the compiler will jump to the program line directly after the FOR instruction. And the loop will run around again.
However, when the value of Loop is incremented and its value is greater than 7, the compiler will allow execution to continue from the line AFTER the NEXT instruction; it will exit the loop. For information, if you were to look at the value in Loop it would actually contain 8 when the loops exits.

STEP
As previously stated, the FOR instruction assumes you mean STEP +1 if you don’t tell it otherwise as this is the most common value.
However, we can ask our loop to count backwards, from 7 to 0 in the next FOR instruction. To do this we must tell the FOR instruction that it is to count backwards and so we specify STEP -1.

If we didn’t, then when the compiler encounters the NEXT instruction, it would increment the Loop variable, making it 8 (remember that the FOR instruction initialised Loop to 7 this time), and 8 is greater than 0, so the loop would only ever be performed once before being exited.

There are two other things worth noting with this program.

You will see that within both loops, the first instruction executed after the FOR instruction, is:

```assembly
Low PORTC ' Force all the PORTC outputs to LOW - All LEDs will now be off
```

In previous BASIC programs we written, we’ve set individual bits of a port LOW or HIGH. Well, if you don’t specify a specific bit, the compiler will assume you mean all the bits and so will affect them all so in this case, all bits of PORTC will be set LOW.

If we didn’t execute this instruction instead of one LED only being on, we would first get one LED, then two, then three and all the way up to eight. Then then program would appear to stop doing anything as all the LEDs would be on and there is no code in the program to switch any of them off.

The other thing worth noting is the “range” of the two loops.
The first loop runs from 0 to 7. When it exists bit 7 of PORTC is still set HIGH.

The second loop then runs from 7 to 0 (counting backwards of course), so it again sets bit 7 of PORTC high. This is a waste of time. You may as well change the range of the second loop from 6 to 0. But once again, when the second loop reaches 0 and exits, the first thing the first loop does, is start counting from 0, and bit 0 is already set, so it would be sensible to actually make the second loop count from 6 to 1.

If you don’t change the second loop to “6 to 1”, and the delay was increased from 50ms to say 1 second, you would see a two second delay each time one of the loops exit, as the next loop would set a specific bit in PORTC that was already set by the previous loop.
We’ve covered rather a lot in this part and you may want to go on and experiment a little on your own.

Try making the Knight Rider display run quickly in one direction, then slower in the other direction.

Add two additional LEDs to the circuit, one connected to PORTB.0 and the other to PORTB.1. When the display is running left to right, illuminate one of the new LEDs, and when the display is running right to left, illuminate the other new LED.