The Queen's University of Belfast

Dept. Electrical & Electronic Engineering

Programmable Integrated Controllers

from Microchip

Written by :

Kieran McCormick and Mark Duncanson

(Electronics Workshop - Ashby Building)

Contents

Introduction

The information Contained in this document has been put together to provide a basic beginners guide to the Microchip's Programmable Integrated Controllers.

This is an introduction only, so not all features will be included. In this document, we have based our work on the PIC16C84, as it is an EEPROM enabling easy re-programming. Programming and re-programming is made very simple with as little external hardware as possible. Most commands are common to all the PICs, although, you should refer to Microchips Databook for finer details

This document should enable the reader to write and test a simple program and then program the PIC16C84 to carry out this operation.

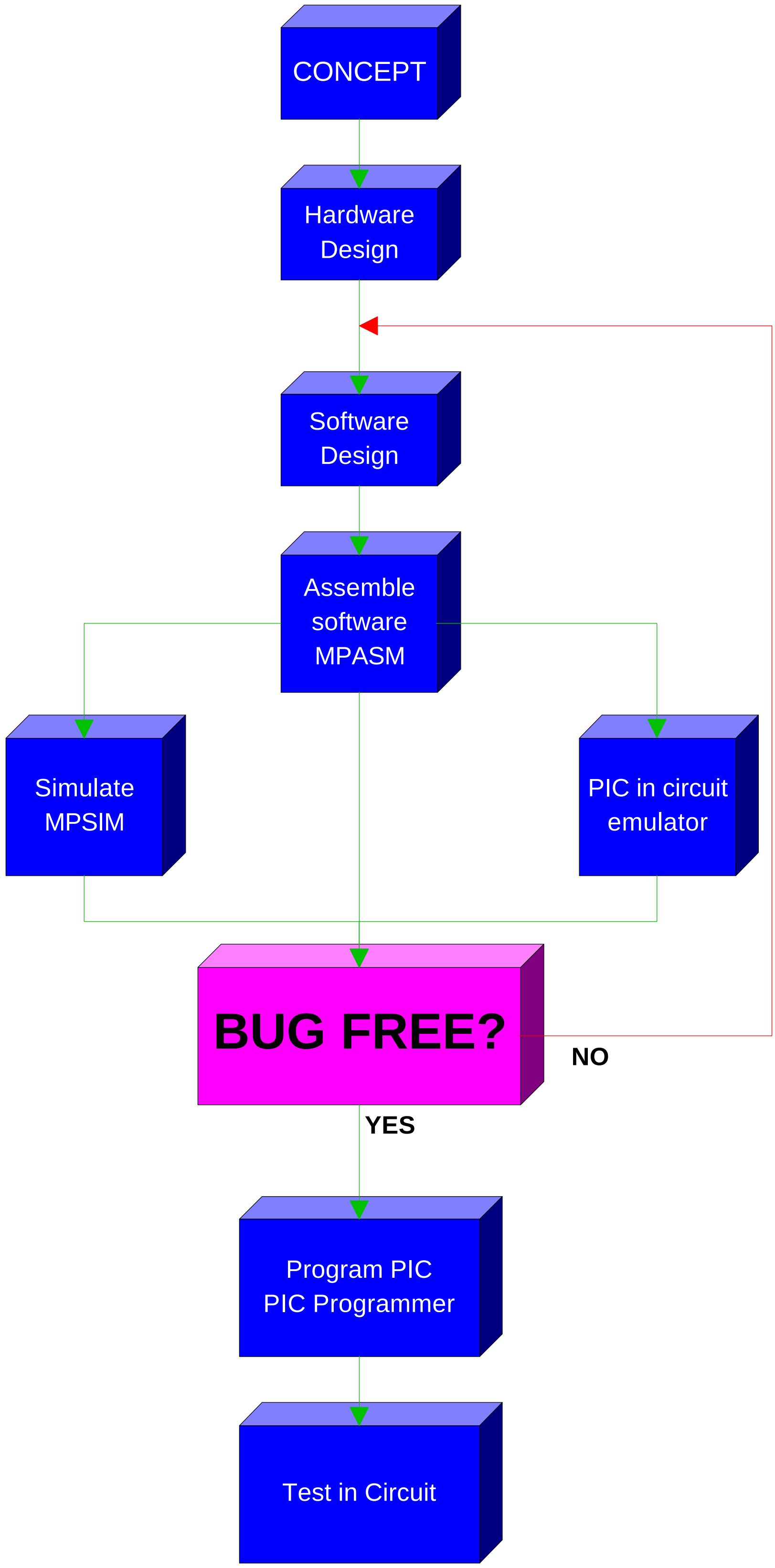
What is the PIC

PIC stands for Programmable Integrated Controller, a complete microcontroller built into an integrated circuit. It can be used for numerous applications, where control is required, whether it be automated or manual. Examples include Traffic Light Controllers and Car Indicator Timers.

The PIC can be programmed, on computer, using assembly language and assembled using MPASM, Arizona Microchip's PIC Assembler. This is to be used to encode the PIC with a programmer connected to your printer port on the computer. When the PIC is programmed, it should be able to control operations external to the computer or programmer, on it's own, providing all hardware is correctly set up.

A simulator program for PICs is also available, MPSIM, which allows you to single step programs while examining the registers and counter, etc., on the screen. This is a vital tool for debugging your program and ensuring that you are satisfied with it's performance. Some PICs are not easily erasable, so it is cost effective to test your program before pre-programming the PIC.

Development Flowchart



Microprocessors and Microcontrollers

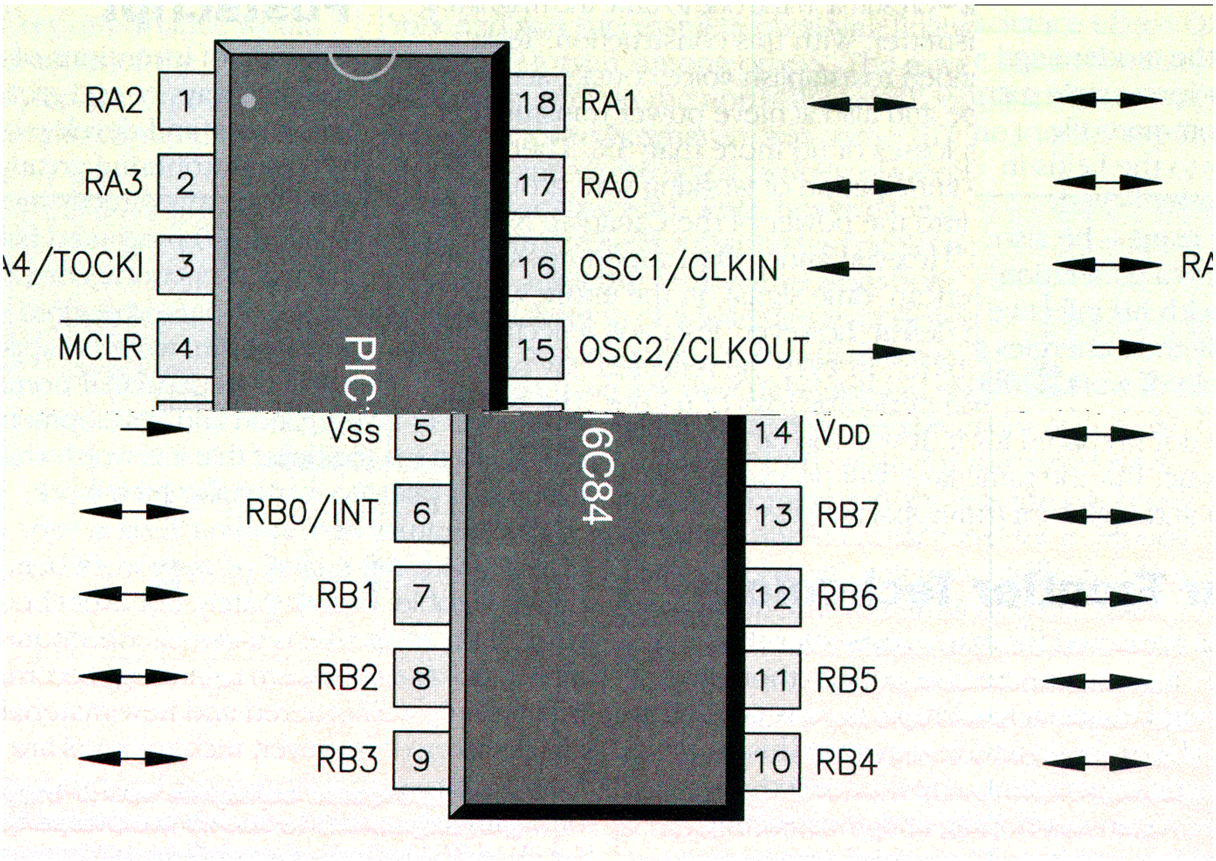
The **microprocessor** is made up of three section:-

Central Processing Unit	Carries out all **calculation** and **data manipulation**.
Input/Output	Used to **communicate** with outside world.
Memory	Stores the **program information**. It can be RAM, ROM, EPROM or EEPROM, depending on how permanent the program needs to be stored.

A **microcontroller** is the complete control system. It houses a microprocessor and other circuitry. It's components are :-

Microprocessor	**Digital computer system** as outlined above.
Oscillator	**Clock data and instruction** into the microprocessor.
Watchdog Timer	**Prevents system latchup**.
Buffering	Allows address and data busses to connect to many chips **without any deteriorating logic levels**.
Decode Logic	Lets I/O or addressing **select** one of a few circuits connected on the same data or address bus.

Pinout and Pin Description



Pin Names	**Description**
RA0 - RA3 -	Lower 4 bits of Port A, Bi-directional (TTL input level)
RA4/TOCK1	5th bit of Port A Open Collector Input/Output (Also clock I/P to TMR0)
RB0/INT	Low bit of Port B, Bi-directional/Ext. Interrupt Input (TTL input level)
RB1 - RB7	Upper 7 bits of Port B, Bi-directional (TTL input level)
$\overline{MCLR}$/Vpp	Master Clear(external reset). Must be kept HIGH for normal operation
Vss	Supply Voltage
Vdd	Supply Ground
OSC1/CLKIN	Clock Input/Oscillator Connection
OSC2/CLKOUT	Oscillator Connection or Clock Out in RC mode (RC is internal)

Commonly Used Commands

As with any programming language, you must learn commands and how to use them properly. The full instruction set for PIC16C84 is on page 2-569 of 1994 Microchip Databook.

Each command line has it's Mnemonic, (the operation to be performed), and in most cases operands, (which are the details of registers or data to be processed.)

The standard commands allow you to manipulate data which may be from memory locations, working register or literal data, (actual numbers in the program). The ability to add and subtract, use logic (AND, OR and XOR), move data between locations and call subroutines are all available.

First we will look at the registers and some of the operand types and what they mean.

f - Register file address (0x00 to 0x7F).

'f' can be any of the memory address locations between 0x00 and 0x7F. It can be given a tag or name by entering an equate line,

register name EQU *register location*

Then rather than having to use the location in commands you may quote the name you have given the register.

W - Working register.

'W' is also known as the accumulator. It is an 8-bit register used for all ALU operations. It is NOT part of the data memory.

k - Literal field, constant data or label.

'k' is used where a literal number is to be used in a command line. It also represents the name of a label, which may be used in a subroutine. You can put a label before a command line when typing it in, so it will then be in the left hand side of the screen. It is good practice to use a TAB before each line, so that you can easily see where labels are. You should also TAB your spaces in the Program File Editor.

d - Destination select; If d=0, store result in W, d=1 store result in f.

'd' can only be '1' representing 'f' or '0' representing 'W'. After an operation has been done, in most cases you have the option of storing the result in the 'W' register or 'f' register, depending on where you want the result stored.

Ex. 1 - Adding the contents of W, the working register, with f, the address of a memory register which is 07H, leaving the result in the working register.

Before instruction W = 0x08 f = 0x04

Command to use - ADDWF f,d Add W and f

Label	Mnemonic	Operand	Data	Comments
	TOTALREG	**EQU**	**07H**	This gives location 07H a tag or name "TOTALREG" (TOTALREG=07H)
	ADDWF	**TOTALREG,0**		Adds 'f' to 'W' and because of the '0' after the comma the result is stored in 'W'

After instruction W = 0x0C f = 0x04

Ex. 2 - Calling a subroutine called DELAY from a different part of the program.

Command to use - CALL k Call subroutine
RETURN Return from subroutine

Label	Mnemonic	Operand	Data	Comments
	COUNT	EQU 08		Gives 08 the tag COUNT
	CALL	**DELAY**		Looks for label DELAY anywhere in the program
	.	.		
	.	.		
	.	.		
DELAY	MOVLW	FF		Puts FF into W register
	MOVWF	COUNT		Puts contents of W into COUNT
LOOP	DECFSZ	COUNT,1		Decrements COUNT leaving result in COUNT (Skip next line if result is Zero)
	GOTO	LOOP		Loop back up to previous line LOOP
	RETURN			Return from subroutine to next instruction after CALL DELAY

More Advanced Commands

Below there are some examples of the use of slightly more unusual commands.

Ex.3 Use the AND command to identify common bits in two different memory registers 11H and 12H. The result is to be stored in memory register 13H.

The contents of one of the registers must be put in the W register so that the ANDWF instruction can be used. Then use the destination for the result as '0' (W register), so as not to change the original memory register. Assume the following contents of 11H and 12H.

Before Instructions	11H = A3	(10100011)
	12H = 7C	(01111100)
AND Result expected	13H =	(00100000) or 20H

Label	Mnemonic	Operand	Data	Comments
REG1	EQU	11H		Name Register 11H with REG1
REG2	EQU	12H		Name Register 12H with REG2
RESULT	EQU	13H		Name Register 13H with RESULT
MOVF	REG1,0			Moves contents of REG1 (A3) into W register (dest. is W indicated by Zero after comma)
ANDWF	REG2,0			Logic AND W reg. with REG2 (7C) (dest. is W indicated by Zero after comma)
MOVWF	RESULT			Moves W contents into RESULT or loc. 13H

After Instructions	11H = A3
	12H = 7C
	13H = 20H

Now let's look briefly at some other useful commands.

Instruction		Description
RLF	f,d	Rotate left register f through carry to destination, d. To rotate within the register f use the number 1 for d.
Example		Before f = 11101101 carry = 0 After f = 11011010 carry = 1
RRF	f,d	Rotate right register f through carry to destination, d.
Example		Before f = 11101101 carry = 0 After f = 01110110 carry = 1
SWAPF	f,d	Swaps halves of the 8 bit register f. Again d represents destination.
Example		Before f = A7 After f = 7A

Byte-oriented Instructions

Instruction	Syntax	Description	Status Affected	Action	Example
BCF	BCF f,b	Bit Clear f	None	Bit b in register f is reset to 0	BCF FLAG_REG, 7 Before Instruction FLAG_REG = 0xC7 After Instruction FLAG_REG = 0x47
BSF	BSF f,b	Bit Set f	None	Bit b in register f is reset to 1	BSF FLAG_REG, 7 Before Instruction FLAG_REG = 0x0A After Instruction FLAG_REG = 0x8A
BTFSC	BTFSC f,b	Bit Test, Skip if Clear	None	If bit b in register f is 0 then the next instruction is skipped. If bit is 0 the next instruction, fetched during current instruction execution, is discarded and a NOP is executed instead making this a two cycle instruction	HERE BTFSC FLAG, 1 FALSE GOTO PROCESS_CODE TRUE... Before Instruction PC = address HERE After Instruction if FLAG<1> =0, PC =address TRUE if FLAG<1> =1, PC =address FALSE
BTFSS	BTFSS f,b	Bit Test, skip if Set	None	If bit b in register f is 1 then the next instruction is skipped. If bit is 1 the next instruction, fetched during current instruction execution, is discarded and a NOP is executed instead making this a two cycle instruction	HERE BTFSS FLAG, 1 FALSE GOTO PROCESS_CODE TRUE... Before Instruction PC = address HERE After Instruction if FLAG<1> =1, PC =address TRUE if FLAG<1> =0, PC =address FALSE

Bit-oriented Instructions

Instruction	Syntax	Description	Status Affected	Action	Example
ADDWF	ADDWF f,d	Add w to f	C, DC, Z	Add the contents of the W register to register f. If d is 0 the result is stored in the W register. If d is 1 the result is stored band in register f.	ADDWF FSR, 0 Before Instruction W = 0x17 FSR = 0xC2 After Instruction W = 0xD9 FSR = 0xC2
ANDLW	ANDLW f,d	AND Variable and W	Z	The contents of the W registers are ANDed with the 8-bit variable k. The result is placed in the W register	ANDLW 0x5F Before Instruction W = 0xA3 After Instruction W = 0x03
ANDWF	ANDWF f,d	AND W and f	Z	AND the W register with register f. If d is 0 the result is stored in the W register. If d is 1 the result is stored back in register f.	ANDWF FSR, 1 Before Instruction W = 0x17 FSR = 0xC2 After Instruction W = 0x17 FSR = 0x02
CLRF	CLRF f	Clear f	Z	The contents of register f are cleared and the Z bit is set	CLRF FLAG_REG Before Instruction FLAG_REG = 0x5A After Instruction FLAG_REG = 0x00 Z = 1

CLRW	CLRW	Clear W Register	Z	W register is cleared. Zero bit (Z) is set.	CLRW Before Instruction W = 0x5A After Instruction W = 0x00 Z = 1
COMF	COMF f,d	Complement f	Z	The contents of register f are complemented. If d is 0 the result is stored in W. If d is 1 the result is stored back in register f	COMF f.d Before Instruction REG1 = 0x13 After Instruction REG1 = 0x13 W = 0xEC
DECF	DECF f,d	Decrement f	Z	Decrement register f. If d is 0 the result is stored in the W register. If d is 1 the result is stored back in register f.	DECF f,d Before Instruction CNT = 0x01 Z = 0 After Instruction CNT = 0x0o Z = 1
DECFSZ	DECFSZ	Decrement f, skip if 0	None	The contents of register f are decremented. If d is 0 the result is placed in the W register. If d is 1 the result is placed back in register f. If result is 0, the next instruction, which is already fetched, is discarded. A NOP is executed instead making it a two-cycle instruction	HERE DECFSZ CNT, 1 GOTO LOOP CONTINUE... Before Instruction PC = address HERE After Instruction if CNT = 0, PC = address CONTINUE if CNT <> 0, PC = address HERE + 1
INCF	INCF f,d	Increment f	Z	The contents of register f are incremented. If d is 0 the result is placed in the W register. If d is 1 the result is placed back in the register f.	INCF CNT,1 Before Instruction CNT = 0xFF Z = 0 After Instruction CNT = 0x00 Z = 1
INCFSZ	INCFSZ f,d	Increment f, skip if 0	None	The contents of register f are incremented. If d is 1 the result is placed back in register f. If the result is 0, the next instruction, which is already fetched is discarded. An NOP is executed instead	HERE INCFSZ CNT, 1 GOTO LOOP CONTINUE... Before Instruction PC = address HERE After Instruction CNT = CNT + 1 if CNT = 0, PC = address CONTINUE if CNT <> 0, PC = address HERE + 1
IORWF	IORWF f,d	Inclusive OR W with f	Z	Inclusive OR the W register with register f. If d is 0 the result is stored in the W register. If d is 1 the result is stored back in register f.	IORWF RESULT, 0 Before Instruction RESULT = 0x13 W = 0x91 After Instruction RESULT = 0x13 W = 0x93
MOVF	MOVF f,d	Move f	Z	The contents of register f is moved to destination d. If d=0 destination is W register. If d=1, the destination is file register f itself. d=1 is useful to test a file register since status flag Z is affected.	MOVF f,d After Instruction W = value in FSR register
MOVWF	MOVWF f	Move W to f	None	Move data from W register to register f.	MOVWF OPTION Before Instruction OPTION = 0xFF W = 0x4F After Instruction OPTION = 0x4F W = 0x4F
NOP	NOP	No Operation	None	No operation	NOP

RLF	RLF f,d	Rotate Left f through carry	C	The contents of register f are rotated 1-bit to the left through the Carry Flag. If d is 0 the result is placed in the W register. If d is 1 the result is stored back in register f.	RLF REG1, 0 Before Instruction REG1 = 11100110 C = 0 After Instruction REG1 = 11100110 W = 11001100 C = 1
RRF	RRF f,d	Rotate Right f through carry	C	The contents of register f are rotated 1-bit to the right through the Carry Flag. If d is 0 the result is placed in the W register. If d is 1 the result is stored back in register f	RRF REG1, 0 Before Instruction REG1 = 11100110 C = 0 After Instruction REG1 = 111001100 W = 01110011 C = 1
SUBLW	SUBLW k	Subtract W from Variable	C, DC, Z	The W register is subtracted (two's complement method) from the 8-bit variable k. The result is placed in the W register	SUBLW 0x02 Before Instruction W = 1 C = ? After Instruction W = 1 C = 1; result is positive. If result is negative C = 0
SUBWF	SUBWF f,d	Subtract W from f	C, DC, Z	Subtract (two's complement method) the W register from register. If d is 1 the result is stored back in register f.	SUBWF REG1, 1 Before Instruction REG1 = 0 W = 1 C = 0; result is negative After Instruction REG1 = FF W = 1 C = 0
SWAPF	SWAPF f,d	Swap f	None	The upper and lower nibbles of register f are exchanged. if d is 0 the result is placed in W register. If d is 1 the result is placed in register f.	SWAPF REG, 0 Before Instruction REG = 0xA5 After Instruction REG = 0xA5 W = 0xA5
XORWF	XORWF f,d	Exclusive OR W with f	Z	Exclusive OR the contents of the W register f. If d is 0 the result is stored in the W register. If d is 0 the result is stored in the W register. If d is 1 the result is stored back in register f.	XORWF REG, 1 Before Instruction REG = 0xAF W = 0xB5 After Instruction REG = 0x1A W = 0xB5

Variable/Control Operations

Instruction	Syntax	Description	Status Affected	Action	Example
ADDLW	ADDLW k	Add Variable to W	C, DC, Z	The contents of the W register are added to the 8-bit variable k and the result is placed in the W register.	ADDLW 0x15 Before Instruction W = 0x10 After Instruction W = 0x25
CALL	CALL k	Subroutine call	None	Subroutine call. First, return address (PC+1) is pushed on to the stack. The 11-bit immediate address is loaded into PC bits 0 to 10. The remaining upper bits of the PC are loaded from PCLATH (f03).	HERE CALL THERE Before Instruction PC = address HERE After Instruction PC = address THERE TOS = address HERE
CLRWDT	CLRWDT	Clear Watchdog Timer	TO, PD	CLRWDT instruction resets Watchdog Timer. It also resets the prescaler of WDT. Status bits are set	CLRWDT Before Instruction WDT counter = ? After Instruction WDT counter = 0x00 WDT prescaler = 0 TO = 0; PD = 0
GOTO	GOTO k	Branch	None	GOTO is an unconditional branch. 11-bit immediate value is loaded into PC bits 0 to 10. Upper PC bits are loaded from bits 3 and 4 of PCLATH. GOTO is a two cycle instruction	GOTO THERE After Instruction PC = address of THERE
IORLW	IORLW k	Inclusive OR Variable with W	Z	The contents of the W register are OR'ed with the 8-bit variable k. The result is placed in the W register	IORLW 0x35 Before Instruction W = 0x9A After Instruction W = 0xBF
MOVLW	MOVLW k	Move Variable to W	None	The 8-bit variable k is loaded into the W register	MOVLW 0x5A W = 0x5A
OPTION	OPTION	Load OPTION register	None	The contents of the W register is loaded into the OPTION register. This instruction is only supported by the PIC16C84	OPTION Before Instruction OPTION = ? After Instruction OPTION = W
RETFIE	RETFIE	Return from Interrupt	None	Return from interrupt. Stack is popped and Top Of Stack (TOS) is loaded in PC. Interrupts are enabled by setting the GIE bit (INTCON register, bit 7). This is a two cycle instruction	RETFIE After Interrupt PC = TOS GIE = 1
RETLW	RETLW k	Return Variable to W	None	The W register is loaded with the 8-bit variable k. The PC is loaded from the top of the stack - the return address. This is a two cycle instruction	RETLW Before Instruction W = ?; PC = ? After Instruction W = k; PC = return address
RETURN	RETURN	Return from Subroutine	None	Return from subroutine. The stack is popped and the top of the stack (TOS) is loaded into the program counter. This is a two cycle instruction.	RETURN After Interrupt PC = TOS
SLEEP	SLEEP	SLEEP Mode	TO, PD	SLEEP mode. Power down status bit (PD) is cleared. Time-out status bit (TO) is set. Watchdog Timer and Prescaler are cleared. Processor is put into SLEEP Mode with clock stopped.	SLEEP
TRIS	TRIS f	Load TRIS register	None	The contents of the W register is loaded into the control register f, where f = 5,6 or 7. This Instruction is only supported by the PIC16C84.	TRIS f Before Instruction f = ? After Instruction f = W
XORLW	XORLW k	Exclusive OR variable with W	Z	The contents of the W register are XOR'ed with the 8-bit variable k. The result is placed in the W register	XORLW 0xAF Before Instruction W = 0xB5 After Instruction W = 0x1A

Development of a Simple Program

Now you can see how to develop a program from an idea.

Example

Two alternately flashing LED's are required for this simple program. The delay between them changing is to be noticeable but not slow.

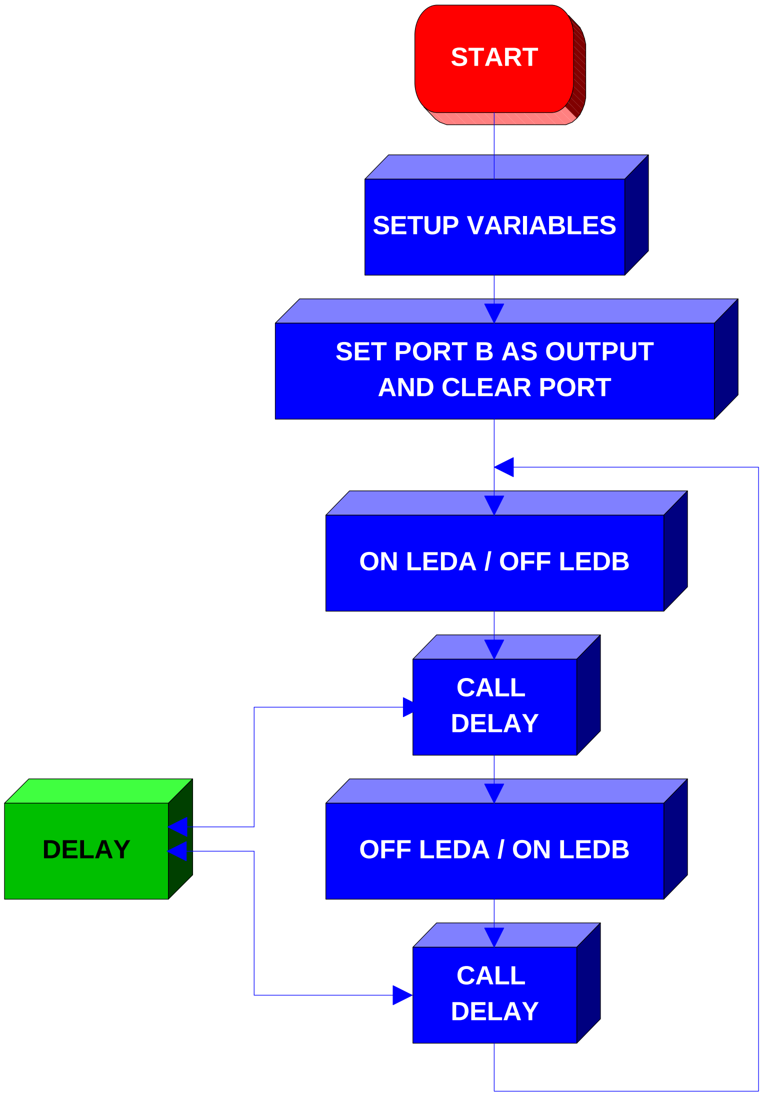
What things are needed to do this?

1. Two separate Bits need to be assigned as outputs from the PIC.
2. A delay of less than 1 second but more than 0.1 seconds is needed.
3. LED A has to be HIGH and LED B has to be LOW, for one DELAY.
4. LED B has to be HIGH and LED A has to be LOW, for one DELAY.
5. Jump back to 3 and keep repeating.

What needs to be set up?

1. PortB bits 0 and 1 can be setup as LED A and LED B respectively.
2. The label PORTB can be put on Location 06H
3. Real Time Clock Counter Register (RTCC) is at 01H
4. Timer Counter, COUNT set to 00H
5. TIME set to 00H

Program Flowchart



Program

```
; Alternating LEDs routine - Filename:   ledonoff.asm
; Stephen Waddington

;Set variables
PORTB           EQU     06H     ;       PORTB is register 6
RTCC            EQU     01H     ;       PIC RTCC timer register
COUNT                   EQU     00H     ;       Timer counter
TIME            EQU     08H     ;       Timer period

; Initialisation routine
INIT    ORG             00H             ;       Store program at location 00H
        TRIS            PORTB           ;       Set PORTB as outputs
        CLRF            PORTB           ;       Clear PORTB

; Main program
MAIN    MOVLW           B'00000001'     ;       Set LEDA on, LEDB off
        MOVWF           PORTB
        CALL            DELAY           ;       Hold LEDA on for .256ms
        MOVLW           B'00000010'     ;       Set LEDB on, LEDA off
        MOVWF           PORTB
        CALL            DELAY           ;       Hold LEDB on for .256ms
        GOTO            MAIN

; Delay routine
DELAY CLRWDT                            ;       Clear Watchdog timer
        MOVLW           TIME
        MOVWF           COUNT
        CLRF            RTCC            ;       Clear RTCC register
LONG    BTFSC           RTCC,7          ;       Test RTCC bit 7(128 x 256us = 32.768us)
        GOTO            JUMP            ;       If RTCC bit 7 set goto JUMP
        GOTO            LONG            ;       If RTCC bit 7 not set then loop until set
JUMP    CLRF            RTCC            ;       If RTCC bit 7 set then clear RTCC
        DECFSZ          COUNT,F         ;       Decrement COUNT by 1 until zero
                                        ;       (32ms x 8 = 0.256s)
        GOTO            LONG            ;       Loop LONG if COUNT <> zero
        RETURN                          ;       Return to call location
RESET GOTO              INIT            ;       On RESET goto INIT

        END
```

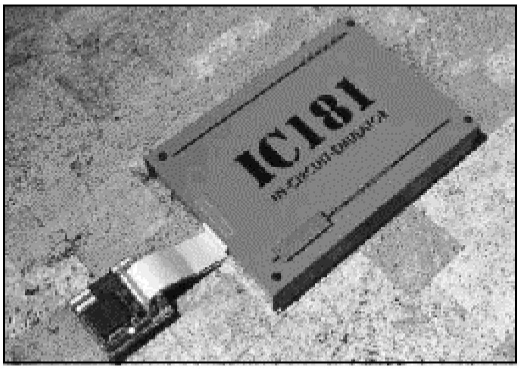
From Source Code to Application

Once you have written the program code for your application it will need to be tested and possibly debugged. This can be done in one of three ways

- In Circuit Emulator (ICE)
- Software Simulation
- Download to PIC

In Circuit Emulator (ICE)

This is a device which plugs into your target circuit and is controlled by the computer and allows real-time testing of the program code. This method carries a cost as in most cases a different ICE is needed for the different families of the PIC and the ICEs start at about approx. £500



Example For An In Circuit Emulator

Software Simulation

MicroChip have produced MPSIM which enables PIC code to be emulated by a PC and various program variables, interrupts, and ports to be monitored.

Download to PIC

The final method is to download the code to the PIC that is intended for final use. This method is the most commonly used as you can try the code in the target circuit in real-time without the expense of an ICE.

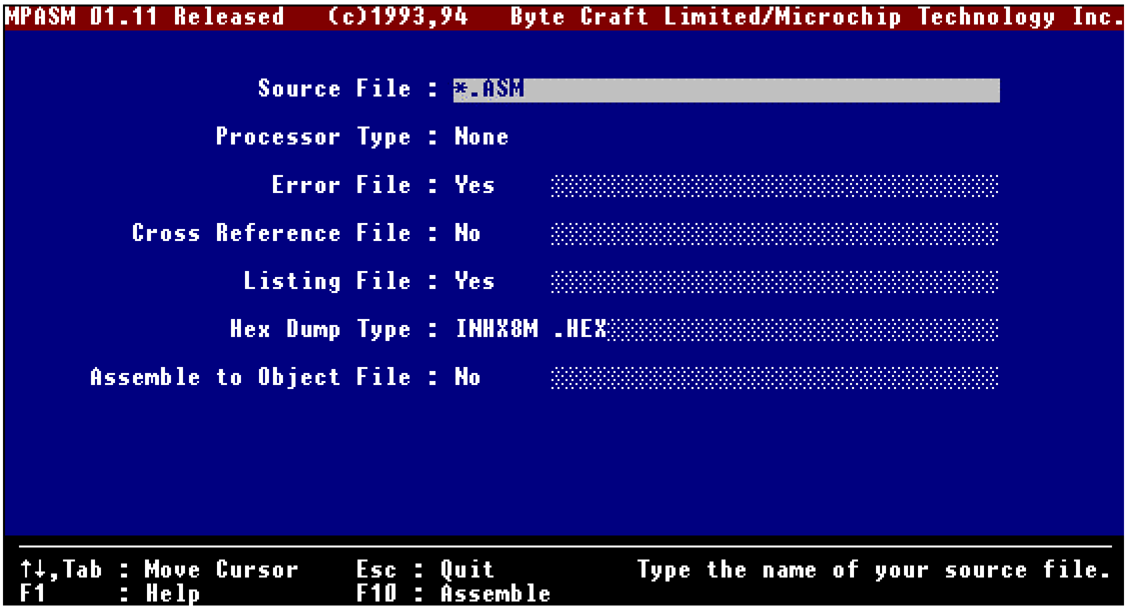
Assembling the Code

Before you can test the code it has to be assembled from its ASCII format file to a hexadecimal format which can be simulated or downloaded. An In Circuit Emulator may need hexadecimal code but some types will work with just the ASCII format code.

There are various assemblers available. The most common is MPASM which is provided by MicroChip, this is needed to create code for MPSIM and various download software. Another assembler produced by MicroChip is MPALC which is required by some download programs.

MPASM

MPASM is a DOS based program that accepts source code in a standard ASCII format and allows the user to select the required output format on screen. It also has a reasonable level of error reporting for when things inevitably don't go right first time. MPASM's main screen looks as follows:



Use of MPASM

1. Enter the name of the source file (Code in ASCII Format) i.e. [ledonoff.asm]
2. Select type of processor i.e. PIC16C84
3. Select the Hex output format i.e. INHX8M (The assembler is able to create four different Hex output formats, depending on the format required by the PIC programmer.)
4. By default the assembler is set to output an Error file and Listing file so to start the assembler simply press [F10]

The assembler returns a series of statistics relating to the length of the length of the assembled code as well as the number of warning and error messages. If the code contains any bugs it will not run when downloaded to the PIC. Errors reported in either the List or Error files. The creates two other in addition to the list and error files. These are as follows:

- **<filename>.asm**
  Default source code file
- **<filename>.lst**
  Default output extension for listing files generated from the assembler

- **<filename>.err**
  Default output extension from MPASM for error details.
- **<filename>.hex**
  Default output code for porting to target PIC
- **<filename>.cod**
  Default output extension for the symbol and debug file.

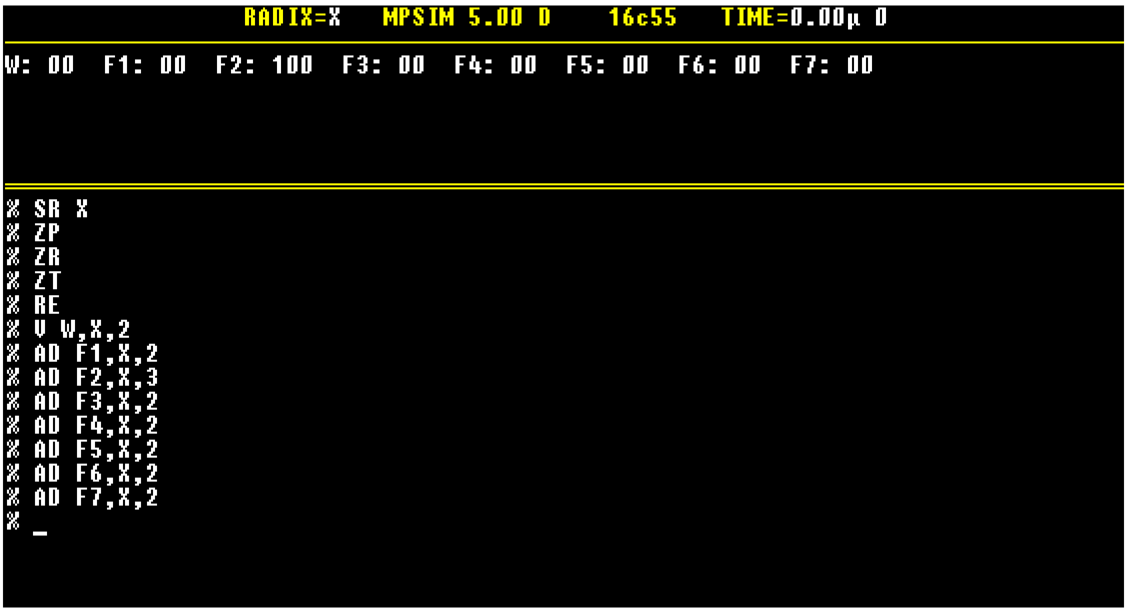
In Circuit Emulator (ICE)

In circuit emulators operate differently depending on the make and model but the basic use is the same:

1. A lead from the ICE is connected to the IC socket on the target circuit where the PIC will be going once programmed.
2. The ICE is connected to the PC, in most cases this is done via the serial communication port.
3. The ICE software is then run on the PC which takes your program code and makes the ICE run like the programmed PIC in real time.

MPSIM

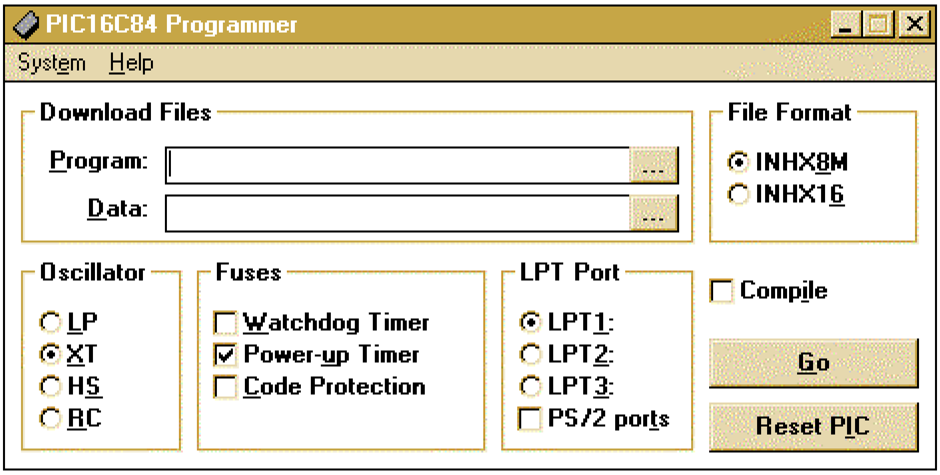
Like the MicroChip assembler MPASM, MPSIM is DOS based and as such, is not very user friendly. It uses a set of proprietary instructions to both initialise the simulator environment and run an actual simulation. It’s almost as if you need to learn an additional software language before you can run a simulation. For this reason, the majority of people tend to test software by downloading it directly to the target PIC. The MPSIM main screen looks as follows:



MPSIM is supplied with a comprehensive user manual in an ASCII format text file.

Download to PIC

Whether for testing or for final production the last stage is to place the program on to the PIC ready for use. There are a number of programming devices available. The programming device consists of two elements, a software program and a programming board. An example of a software program is shown below:

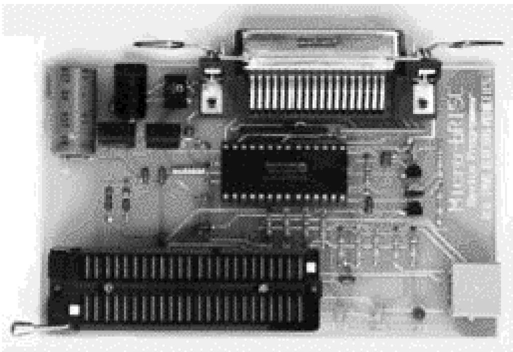


The software is used to drive the programming board which in most cases holds the PIC for programming in a zero insertion force (ZIF) socket.
Programming a PIC is Very straightforward. Having load the programmer application and connected the programming board the following information is selected.

- Program source code. (i.e. ledonoff.hex)
- The file format that the source code is in (i.e. INHX8M)
- The type of oscillator being used on the target circuit
- The PICs software fuses that need to be set

Once this is complete, the target device is mounted on the programming board and the code downloaded. It takes approximately 20 to 30 seconds to program a PIC device. During this time the PC transfers the hex code and fuse selections to the memory of the PIC, before verifying the contents of all EPROM or EEPROM memory.

The PIC can now be removed from the programmer and placed in the target circuit ready for use.



Example Of a PIC Programmer Board

Appendix (A)
Example Applications

Traffic Light Sequencer

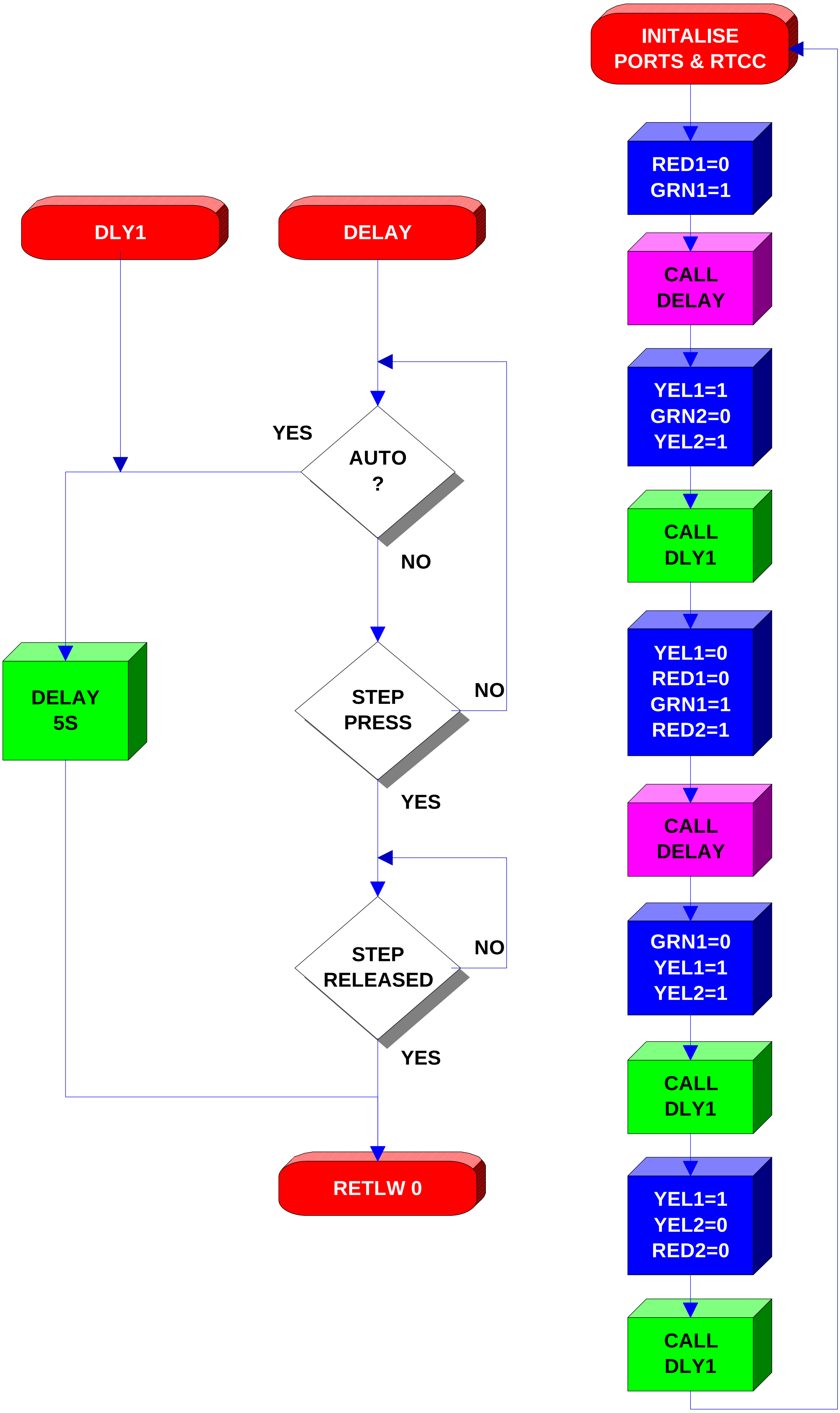
This traffic light sequencer could be used as part of a model railway set or with slight modification be scaled up for roadside use.

The basic unit steps through its sequence either manually or at a fixed speed. For home use, the speed of 5 seconds between step is probably adequate. In the auto mode, the lights continually sequence. The manual mode changes the lights from one direction of traffic flow to the other enabling the standard ‘road work’ operation or a 4 way junction.

To enable the design to be used commercially, a variable time control could be added to change the duration between changeover sequences. The design would then run on a 16C71, 73 or 74 and the various time settings in an analog form could be read and converted to actual times within the software.

The light sequence is set in the software and follows the standard “Red - Yellow - Green - Green & Yellow - Red” pattern. Modification of the software for 3 way junctions can easily made by increasing the number of blocks of light patterns with their associated delays.

Traffic Light Sequencer (program flowchart)

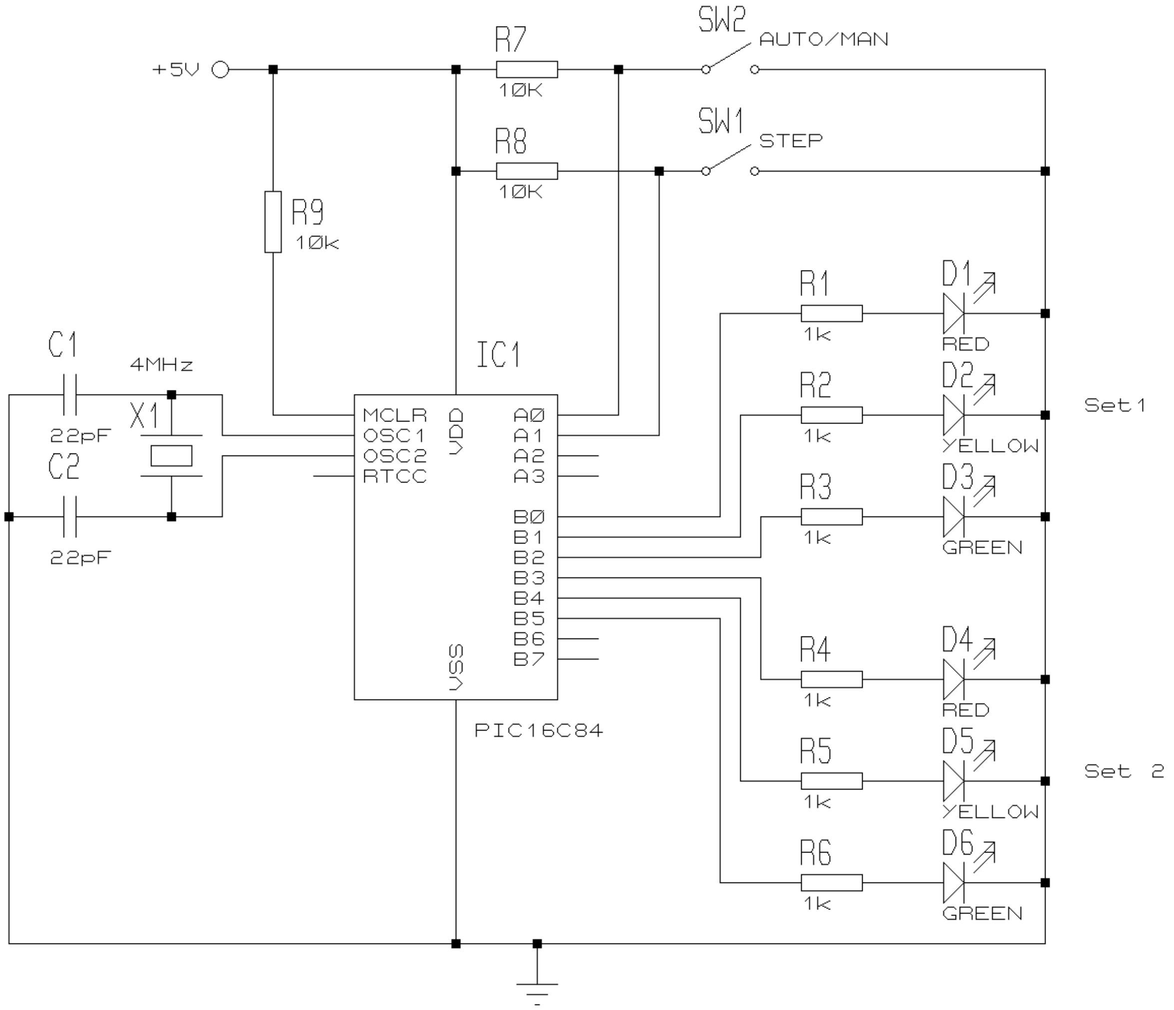


Traffic Light Sequencer (source code)

```
; WRITTEN BY                NIGEL GARDNER
; COPYRIGHT                 BLUEBIRD ELECTRONICS
; DATE                      21/2/95
; ITERATION                 1.0
; FILE SAVED AS             TRAF1.ASM
; FOR PIC16C54
; 4.00 MHz RESONATOR.
; INSTRUCTION CLOCK         1.00 MHz  T= 1uS

; SOFTWARE WRITTEN FOR USE WITH PROJECT BOARD FROM BLUEBIRD
; ELECTRONICS.

; ***** EQUATES *****

RTCC        EQU     1       ; COUNTER
PC          EQU     2       ; PROGRAM COUNTER
STATUS      EQU     3       ; STATUS REGISTER
CARRY       EQU     0       ; CARRY BIT
DCARRY      EQU     1       ; DIGIT CARRY BIT
PDOWN       EQU     3       ; POWER DOWN BIT
WATDOG      EQU     4       ; WATCHDOG TIMEOUT BIT
FSR         EQU     4       ; FILE SELECT REGISTER
Z           EQU     2

TIME        EQU     .156    ; 156 * 64mS = 10 SECONDS

PORTA       EQU     5
AUTO        EQU     0       ; MANUAL AUTO SWITCH
STEP        EQU     1       ; SEQUENCE STEP SWITCH

PORTB       EQU     6
RED1        EQU     0       ; SET A OF LIGHTS
YEL1        EQU     1
GREEN1      EQU     2
RED2        EQU     3       ; SET B OF LIGHTS
YEL2        EQU     4
GREEN2      EQU     5

COUNT       EQU     0BH     ; GENERAL PURPOSE COUNTER

            ORG     00

; *********** INITALISE PORTS AND RTCC *************

INIT    MOVLW       00H
        TRIS        PORTB           ; PORT B AS OUTPUTS
        CLRF        PORTB
        MOVLW       0FH
        TRIS        PORTA           ; PORT A AS INPUTS
        MOVLW       B'00000111'     ; RTCC PRE-SCALAR /256
        OPTION                      ; 256uS PER COUNT INTERNAL CLOCK

; ********* PROGRAM BEGINS HERE *********************

MAIN    BSF         PORTB,RED1
        BSF         PORTB,GREEN2
        MOVLW       TIME            ; DELAY TIME
        CALL        DELAY
```

```
        BSF     PORTB,YEL1
        BCF     PORTB,GREEN2
        BSF     PORTB,YEL2
        MOVLW   TIME            ; DELAY TIME
        CALL    DELAY

        BCF     PORTB,YEL1
        BCF     PORTB,RED1
        BSF     PORTB,GREEN1
        BSF     PORTB,RED2
        MOVLW   TIME            ; DELAY TIME
        CALL    DELAY

        BCF     PORTB,GREEN1
        BSF     PORTB,YEL1
        BSF     PORTB,YEL2
        MOVLW   TIME            ; DELAY TIME
        CALL    DELAY

        BCF     PORTB,YEL1
        BCF     PORTB,YEL2
        BCF     PORTB,RED2
        GOTO    MAIN

; TEST HERE TO SEE IF MANUAL MODE OR DELAY IF AUTOMATIC

DELAY   BTFSC   PORTA,AUTO      ; TEST FOR AUTO SWITCH ON
        GOTO    DLY1            ; AUTOMATIC MODE
LP1     BTFSC   PORTA,STEP      ; IF MANUAL MODE, THEN WAIT FOR
        GOTO    DELAY           ; BUTTON PRESS BUT CHECK IF AUTO
LP2     BTFSS   PORTA,STEP      ; AND THEN RELEASE BEFORE CONTINUING
        GOTO    LP2             ; TO NEXT SEQUENCE
        RETLW   0

; LONG DELAY 64mS * VALUE IN W REGISTER

DLY1    CLRF    RTCC
        MOVWF   COUNT           ; USE THIS REGISTER TEMPORARILY
LONG2   BTFSC   RTCC,7          ; TEST RTCC BIT 7 64*256uS = 64mS
        GOTO    JMP1
        GOTO    LONG2           ; LOOP UNTIL BIT IS SET

JMP1    CLRF    RTCC            ; YES, SO CLEAR RTCC
        DECFSZ  COUNT,F         ; DECREMENT, UNTIL ZERO
        GOTO    LONG2
        RETLW   0

        ORG     1FFH            ; RESET VECTOR FOR C54
        GOTO    INIT
        END
```

Traffic Light Sequencer (circuit diagram)



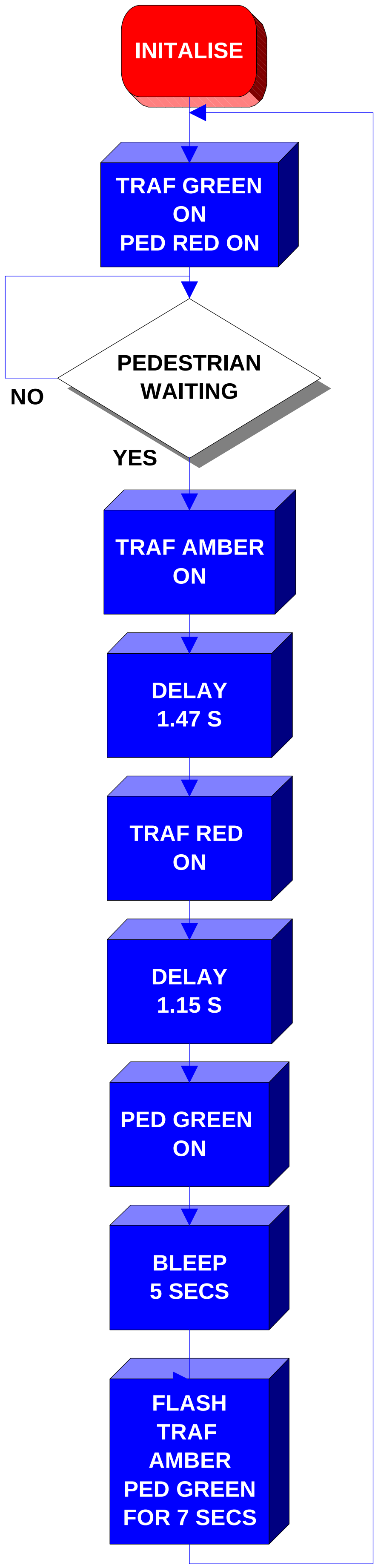
Pedestrian Crossing Simulator

This code is similar to the traffic light sequencer in that it follows the sequence f change from one set of lights to another. However, the addition of sound and flash operation enables this design to become a fully working product with minimal change.

The sounder shown in the diagram is a small loudspeaker. The warning tone frequency is set within the software.

Modification to the design could be to include a delay between cycles to allow sensible traffic flow or the provision of a vehicle sensor to allow faster response times when no vehicles are present.

Pedestrian Crossing Simulator (program flowchart)



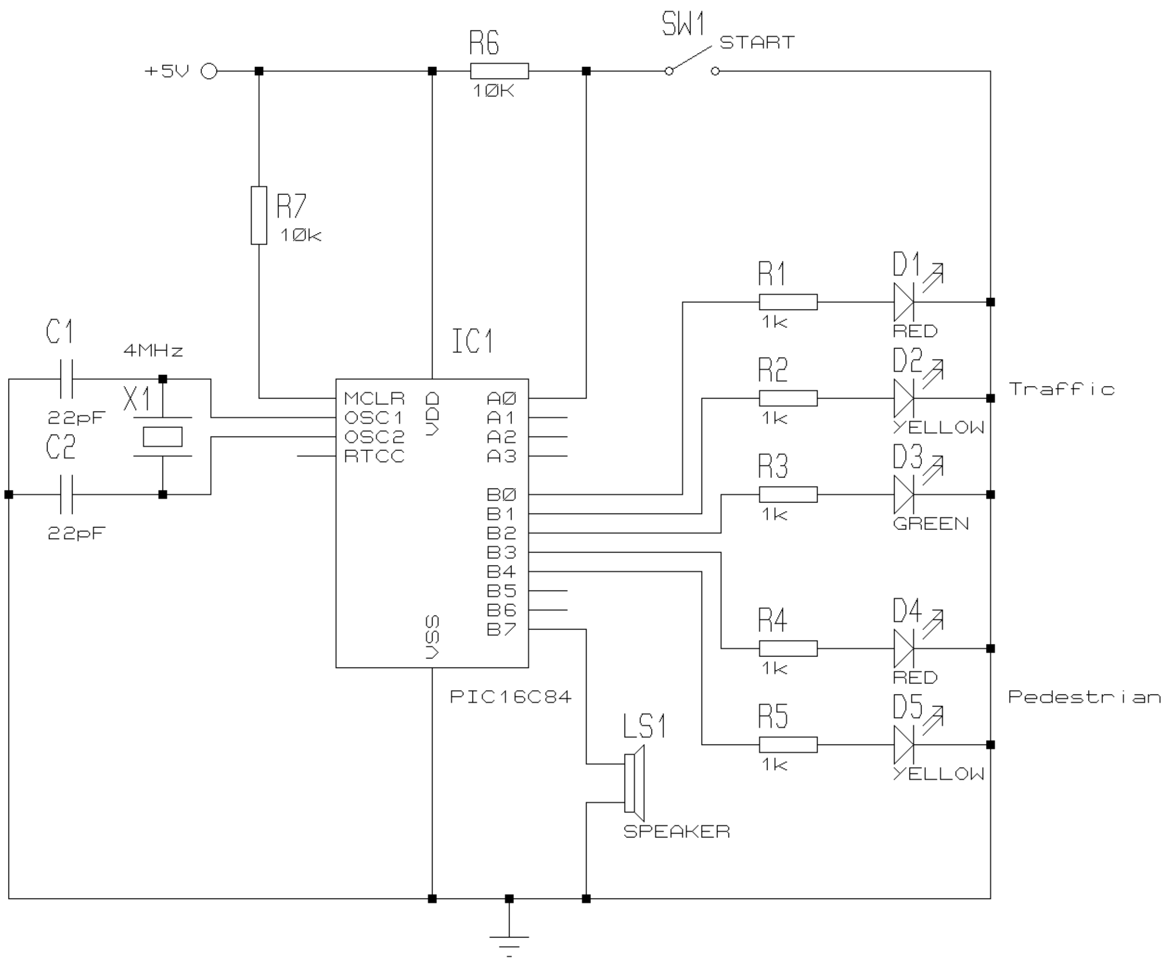
Pedestrian Crossing Simulator (source code)

```
; WRITTEN BY              NIGEL GARDNER
; COPYRIGHT               BLUEBIRD ELECTRONICS
; DATE                    2/8/95
; ITERATION               1.0
; FILE SAVED AS           PED1.ASM
; FOR PIC16C54
; 4.00 MHz RESONATOR.
; INSTRUCTION CLOCK       1.00 MHz  T= 1uS

; Software will run with project board from Bluebird Electronics

; ***** equates *****

rtcc        equ     1           ; counter
pc          equ     2           ; program counter
status      equ     3            ; status register
carry       equ     0           ; carry bit
dcarry      equ     1           ; digit carry bit
pdown       equ     3           ; power down bit
watdog      equ     4           ; watchdog timeout bit
fsr         equ     4           ; file select register
z           equ     2

time2       equ     .200        ; 200*512us = 0.1024S

porta       equ     5
#define     go      porta,0     ; start button

portb       equ     6
#define     red1    portb,0     ; traffic lights
#define     yel1    portb,1
#define     grn1    portb,2
#define     red2    portb,3     ; pedestrian lights
#define     grn2    portb,4
#define     buzz    portb,7     ; buzzer for warning

count       equ     0ch         ; general purpose counter
sound       equ     0dh
flash       equ     0eh

            list p=16c54        ; processor type

            org     00

; ****************** subroutines ************************

; ********* long delay 32ms * value in w register *********

delay1      clrf    rtcc
            movwf   count       ; use this register temporarily
long2       btfsc   rtcc,7      ; test rtcc bit 7 (128*256us = 32.768ms)
            goto    jmp1
            goto    long2       ; loop until bit is set

jmp1        clrf    rtcc        ; yes, so clear rtcc
            decfsz  count,f     ; decrement, until zero
            goto    long2
            retlw   0

; ***** delay with sounder of 1.95KHz ************
```

```
delay2          movlw   time2    ; load timer
dly2            clrf    rtcc
                movwf   count    ; use this register temporarily
                bsf     buzz
long3           btfsc   rtcc,1   ; test rtcc bit 1 (2*256us = 512us)
                goto    jmp2
                goto    long3    ; loop until bit is set
jmp2            bcf     buzz
                clrf    rtcc     ; yes, so clear rtcc
                decfsz  count,f  ; decrement, until zero
                goto    long3-1
                retlw   0

; *********** initalise ports and rtcc *************

init            clrf    portb    ; clear port
                movlw   00h
                tris    portb    ; port b as outputs
                movlw   0fh
                tris    porta    ; port a as inputs
                movlw   b'00000111'      ; rtcc pre-scalar /256
                option                   ; 256us per count internal clock

; ********* program begins here *********************

main            bsf     grn1     ; traffic on green
                bcf     red1
                bsf     red2     ; pedestrian on red
                bcf     grn2
                btfsc   go
                goto    $-1      ; loop for start button
                bcf     grn1
                bsf     yel1     ; traffic on amber
                movlw   .45      ; delay time (1.47S)
                call    delay1
                bsf     red1     ; traffic on red - wait for them to stop
                bcf     yel1
                movlw   .35      ; delay time (1.15S)
                call    delay1
                bcf     red2
                bsf     grn2     ; pedestrian on green
                movlw   .30      ; tone bursts approx 5 secs
                movwf   sound

bleep           movfw   sound    ; reload for next bleep
                call    delay2   ; 0.1S tone burst
                movlw   2        ; delay time
                call    delay1   ; 65mS quiet period
                decfsz  sound,f  ; count down time
                goto    bleep    ; make sound again

                movlw   .9       ; flash for approx 7 secs
                movwf   flash

get_off         movfw   flash    ; reload for next time
                movlw   .12      ; 0.393S off
                call    delay1
                bcf     red1     ; turn off traffic red
                bcf     yel1
                bcf     grn2
                movlw   .12      ; 0.39S on
                call    delay1
                bsf     yel1     ; flash traffic amber
                bsf     grn2     ; flash pedestrian green
```

```
decfsz  flash,f   ; count down
goto    get_off   ; get off xing
bcf     yel1      ; turn off traffic amber
goto    main      ; go again


org     1ffh      ; reset vector for c54
goto    init

end
```

Pedestrian Crossing Simulator (circuit diagram)



Book List

MicroChip DataBooks available from Electronics Store Rm. 10.4 (PIC Related)

- A Beginners Guide to the Microchip PIC (Nigel Gardner)
- PIC Cookbook - Vol. 1 (Nigel Gardner + Peter Birnie)
- PIC16/17 Microcontroller Databook
- Embedded Control Databook 1994/95