

COVENTRY UNIVERSITY
Faculty Of Engineering & Computing

Introduction to the Microchip MCC18 C Compiler

The following is a brief introduction to the Microchip MCC18 C compiler. It assumes that the user is familiar with C and is only used here in explaining how the user can successfully compile and download an object file to the MPLAB simulator and the PIC18 target board. Links to a full manual for the MCC18 compiler and literature on its libraries are given at the end of this document.

Introduction

In our previous work on the PIC18 we looked at the techniques involved in using the MPASM assembler in creating the hex files required for simulating our code. In this session we will use the MCC18 C compiler for a similar task. The overall aim in using both programming environments Assembly or C is to produce the final **.hex** file that we can use to simulate or program our target device. One of the main concerns when creating the code is the size of the hex file. This is for two main reasons. The first reason is to do with the size of available memory on chip. The second reason is code efficiency in code execution.

Assembly language code development produces a hex file that is much smaller compared to the hex file produced by a C compiler. On the other hand C programming is less time consuming and a lot easier to write.

Some of the advantages in using C as the development platform as compared to an Assembly environment are as follows:

- It is easier to and quicker to write in C than Assembly
- You can use code already written and available in function libraries
- Your C source files are portable to other micros with only little changes if any
- It is easier to modify and update

There are several C compilers available that support the PIC18 family of micros. Some of the most popular ones are:

- MCC18 C compiler (from the makers of the PIC18 Microchip)
- HI-TECH C compiler
- CCS C compiler
- mikroC C compiler
- IAR C/C++ compiler

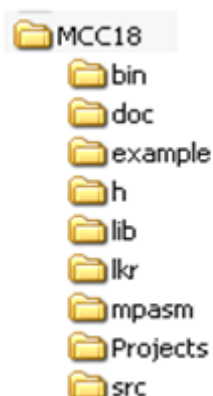
The Microchip MCC18 C compiler is one of the most popular C compilers. There is an academic version of the compiler available for student use.

Features

- ANSI '89 compatibility
- Integration with the MPLAB IDE for easy-to-use project management and source-level debugging
- Generation of relocatable object modules for enhanced code reuse
- Compatibility with object modules generated by the MPASM assembler, allowing complete freedom in mixing assembly and C programming in a single project
- Transparent read/write access to external memory
- Strong support for inline assembly when total control is absolutely necessary
- Efficient code generator engine with multi-level optimization
- Extensive library support, including PWM, SPI, I2C, UART, USART, string manipulation and math libraries
- Full user-level control over data and code memory allocation
- Supports both a small (16-bit pointers) and a large (24-bit pointers) memory model for efficient use of memory
- MPLIB allows easy use of included libraries and for user created libraries
- Extensive multi-pass optimizations
- Supports new PIC18F extended mode instructions

The MCC18 Directory structure

MCC18 is installed on your D: applications drive of your PC. The following directory structure applies:



Directory structure of the MCC18
C compiler

MCC18 Directory structure	
Directory	Description
bin	Contains the executables for the compiler and linker. These are described in more detail in the following section.
cpp	Contains the source code for the MPLAB C18 C preprocessor. This source code is provided for general interest.
doc	Contains the MPLAB C18 electronic documentation. Refer to these documents for questions regarding MPLAB C18.
example	Contains sample applications to help users get started using MPLAB C18, including the examples contained in this document.
h	Contains the header files for the standard C library and the processor-specific libraries for the supported PICmicro MCUs.
lib	Contains the standard C library (clib.lib or clib_e.lib), the processor-specific libraries (p18xxx.lib or p18xxx_e.lib, where xxx is the specific device number) and the startup modules (c018.o, c018_e.o, c018i.o, c018i_e.o, c018iz.o, c018iz_e.o).
lkr	Contains the linker script files.
mpasm	Contains the command-line version of the MPASM assembler, the assembly header files for the devices supported by MPLAB C18 (p18xxx.inc) and the assembly header files used by the libraries.
src	Contains the source code, in the form of C and assembly files, for the standard C library, the processor-specific libraries and the startup modules.

Table 1.

MCC18 Directory structure

MCC18 C Compiler executables (Bin Directory)

Executable	Description
mcc18.exe	This is the compiler shell. It takes as input a C file (i.e., <i>file.c</i>) and invokes the Extended or Non-extended mode compiler executable.
mcc18-extended.exe	This is the Extended mode compiler executable. It is invoked by the compiler shell when compiling for Extended mode. It invokes the preprocessor <i>cpl18.exe</i> to preprocess the C file and then compiles the preprocessed output and generates a COFF file (e.g., <i>file.o</i>) to be passed to the linker.
mcc18-traditional.exe	This is the Non-extended mode compiler executable. It is invoked by the compiler shell when compiling for the Non-extended mode. It invokes the preprocessor <i>cpl18.exe</i> to preprocess the C file and then compiles the preprocessed output and generates a COFF file (e.g., <i>file.o</i>) to be passed to the linker.
cpl18.exe	This is the C preprocessor.
mplink.exe	This is the driver program for the linker. It takes as input a linker script, object files and library files and passes these to <i>_mplink.exe</i> . It then takes the output COFF file from <i>_mplink.exe</i> and passes it to <i>mp2cod.exe</i> and <i>mp2hex.exe</i> .
_mplink.exe	This is the linker. It takes as input a linker script (e.g., <i>p18f452.lkr</i>), object files and library files and outputs a COFF executable (e.g., <i>file.out</i> or <i>file.cof</i>). This COFF file is the result of resolving unassigned addresses of data and code of the input object files and referenced object files from the libraries. <i>_mplink.exe</i> also optionally produces a map file (e.g., <i>file.map</i>) that contains detailed information on the allocation of data and code.
mp2cod.exe	This is the COFF to COD file converter. The COD file is a symbolic debugging file format which is used by the MPLAB IDE v5.xx. <i>mp2cod.exe</i> takes as input the COFF file produced by <i>_mplink.exe</i> and outputs a COD file (e.g., <i>file.cod</i>). It also creates a listing file (e.g., <i>file.lst</i>) that displays the correspondence between the original source code and machine code.
mp2hex.exe	This is the COFF to hex file converter. The hex file is a file format readable by a PICmicro programmer, such as the PICSTART Plus or the PRO MATE II. <i>mp2hex.exe</i> takes as input the COFF file produced by <i>_mplink.exe</i> and outputs a hex file (e.g., <i>file.hex</i>).
mplib.exe	This is the librarian. It allows for the creation and management of a library file (e.g., <i>file.lib</i>) that acts as an archive for the object files. Library files are useful for organizing object files into reusable code repositories.
mpasm.exe	This is the command-line assembler. It takes as input an assembly source file (e.g., <i>file.asm</i>) and outputs either a COFF file (e.g., <i>file.o</i>) or a hex file and COD file (e.g., <i>file.hex</i> and <i>file.cod</i>). It also creates a listing file (e.g., <i>file.lst</i>) and an error file (e.g., <i>file.err</i>), which contains any errors or warnings emitted during the assembly process. Assembly source files may include assembly header files (e.g., <i>p18f452.inc</i>), which also contain assembly source code.

Table 2 Compiler Executables

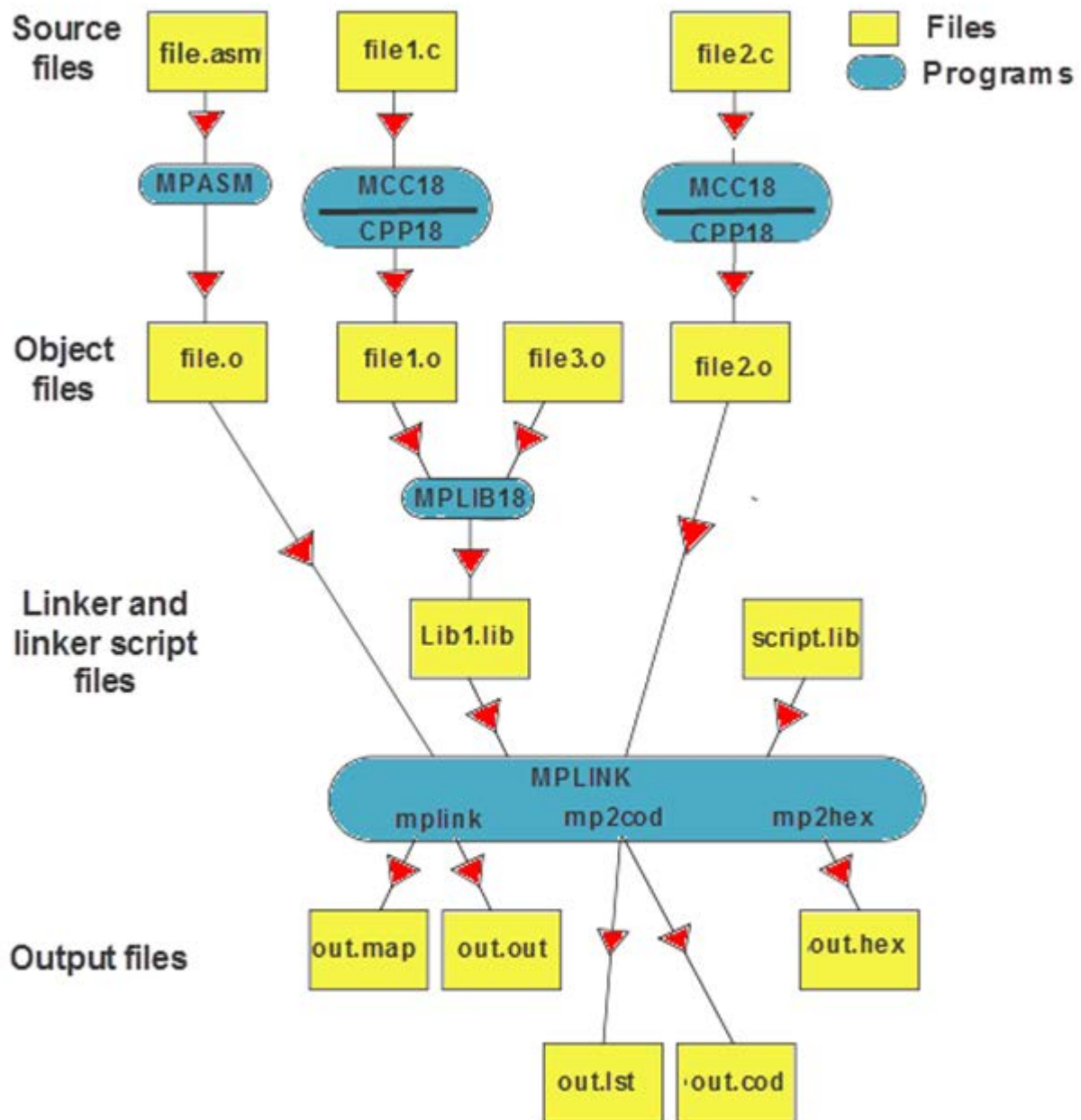


Fig. 1 Execution Flow Of Language Tools

The Compiler source files

The compiler organization is in the form of a **Project** folder. All the required project files are stored in this project folder. As we can see from Fig.1 (Execution Flow of Language Tools) diagram a project file consists of a number of files. These could be one or more C source files as shown; file1.c, file2.c (extension.C). In addition assembler files could be included in the project with the extension.asm. The compilation process will combine all these source files and will generate on successful compilation the object file with the extension .o. The linker will then use device specific libraries together with the object files in order to generate the output files. This is the .hex file that can be loaded into our target micro, the .map file giving information about program and data memory usage, the .lst file which the listing file containing assembly listing of our code, the .cod file which is the symbol and debug file.

The structure of an MCC18 source file

The basic template structure of an MCC18 C source program is as shown in the following table:

```
/* Sample source program for the MCC18 compiler */
/* File name: */
/* Version: */
/* Author: */
/* Company: */
/* Date: */
/*
/* Program function */

/* Include files */
#include <p18f4520.h>
#include .....
#include .....
/*
/* Symbol Definitions */
#define .....
#define .....
.....
#pragma config                /* Microcontroller fuse configuration */

/* Global Variables */
int 1,2,3 ....
Char a,b,c
.....
/* Function Declarations */
int Func(char a)
{
    .....
    .....
    return ....
}
....
.....
/* Main Program starts here */
void main(void)
{
    .....
    ..... Body of the main program in here
    .....
}
```

An example showing a C source file is as shown below. This is the C version of the assembly source file we used in our previous laboratory session under the heading of **MPLAB IDE Introduction**.

```

/* Sample source program for the MCC18 compiler */
/* File name:                                     */
/* Version:                                       */
/* Author:                                       */
/* Company:                                       */
/* Date:                                         */
/* Program function: The following example uses PORT C as a digital output */
/* LEDs attached to the Port will show a toggling values of 1s and 0s      */

/* The following configure operational parameters of the PIC                */

#pragma config OSC = HS      //set osc mode to HS  high speed clock
#pragma config WDT = OFF     // set watchdog timer off
#pragma config LVP = OFF     // Low Voltage Programming Off
#pragma config DEBUG = OFF   // Compile without extra Debug compile Code

/* Include Files                                                            */

#include <p18f4520.h>           // Device used is the PICF4520
#include <delays.h>            // Include the delays routines

unsigned char C = 0xAA;       // Assign hex value AA to C
unsigned char D = 0x55;       // Assign hex value of 55 to D

void main (void)

{

/*Set PORT C as Digital Outputs      */

LATC = 0x00;    // Initialise Port C
TRISC = 0x00;   // Configure Port C as O/P
while (1)       // run forever

{
LATC = C;       // write the value of C (AA) into Port C
Delay10KTCYx(100); // Short delay here so we can see output
LATC = D;       // write the value of D (55) to Port C
Delay10KTCYx(100); // Short delay here

}

}

```

Example of C code

Comments

Let us now examine in a bit more detail the structure of our C source file.

At the beginning of the program we have comments that describe the program function, file name and version, the author/company name and date. This is not necessary information but it provides for good program documentation. As we can see these lines are preceded by the “**/***” characters specifying that they are **comments**. These characters are also used to specify the end of the comment. These are known as **long comments**.

Comments are ignored by the compiler but they are necessary in order to clarify the various program statements in our source file and make our code easy to read and understand.

Comments can also be used after statements to explain the operation performed. These are the **short comments** starting with the characters “**//**” and they do not need to be terminated by another character.

The #pragma statement

This statement is often used to declare data or code sections, declare interrupt vectors and other implementation dependent features. The statement **#pragma config** is used to select the various fuses that control operational features of the micro such as clock source, code protection, low voltage programming, debug etc.

The #include statement

The **#include** statement “includes” other source files that need to be in our source file. The format of the statement is **#include** followed by the filename to be included. In our example the statement:

```
#include <p18f4520.h> ,
```

includes the header file that specifies the names of the various Special Function Registers (SFRs) for our selected micro PIC18F4520..

Our example also contains the statement **#include <delays.h>**. This is a header file for the MCC18 delays library which provides functions to be used in creating set time delays in our program. Further down our main program we will see the statement: **Delay10KTCYx(100)**; this is a function call to a function in the **delays** library. This function creates a time delay with software of 1000 (1K) instruction cycles (TCY) times the argument value. In our example the argument is 100 so this function will create a delay for 100x1000=100,000 instruction cycles. The instruction rate on the PIC18F4520 is ¼th of the oscillator clock i.e. it takes 4 clocks to execute an instruction. If the micro clock is 1MHz the instruction rate is 250 kHz the total delay will be 100,000x4ms=400ms. We will experiment later on in selecting various delay values when displaying flashing LEDs on the IO ports of our target micro.

Variable Types

MCC18 supports the variable types shown in Table 3.

The statements **unsigned char C = 0xAA** and **unsigned char D = 0x55** assign the hex values of AA and 55 to C and D.

unsigned char: are 8-bit unsigned variables with the range of 0-255.

MCC18 Variable types

Types	Size (bits)	Range
char	8	−128 to +127
signed char	8	−128 to +127
unsigned char	8	0 to 255
int	16	−32 768 to +32 767
unsigned int	16	0 to 65 535
short	16	−32 768 to +32 767
unsigned short	16	0 to 65 535
short long	24	−8 388 608 to +8 388 607
unsigned short long	24	0 to 16 777 215
long	32	−2 147 483 648 to +2 147 483 647
unsigned long	32	0 to 4 294 967 295
float	32	±1.17549E-38 to ±6.80565E38
double	32	±1.17549E-38 to ±6.80565E38

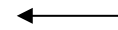


Table 3
Shows MCC18 variable types

We finally have the code of the main program that starts with the statement:

void main (void) and the body of the main program is contained by the curly brackets “ {} ”

LATC: This variable is used to address PORT C *latch register* and set the output state of RC0-RC7 pins. The instruction **LATC = 0x00;** initializes Port C to 0s .

TRISC: This variable is used to access the **S**pecial **F**ile **R**egister used to configure the Port C operation as input or output. TRISC stands for *Tr*istate register C. By setting the contents of this register to 1s and 0s we can select the data direction of the port. A TRIS bit value of 0 configures the associated port pin to output while a bit value set to 1 will configure the corresponding pin to input. The statement **TRISC = 0x00;** configures all pins of PortC as outputs.

while (1): An infinite while loop is used here to keep the micro running and stop it from exiting **main** and therefore executing unknown code by accessing undefined memory locations.

LATC = C; statement writes the value of C (0xAA) into the Latch register of Port C. If we were to examine the pins of Port C we will see logic states as shown in Fig.2 :

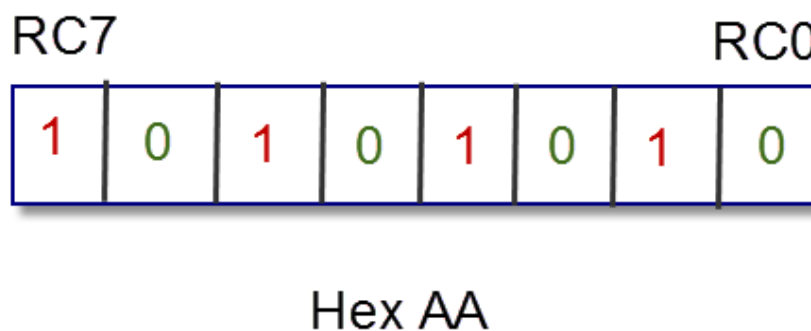


Fig.2
Contents of LATC
0xAA

Delay10KTCYx(100); as explained above is a function call to provide a time delay for our Port C output

LATC = D; statement writes the value of D (0x55) into the Latch register of Port C. If we were to examine the pins of Port C we will see logic states of :

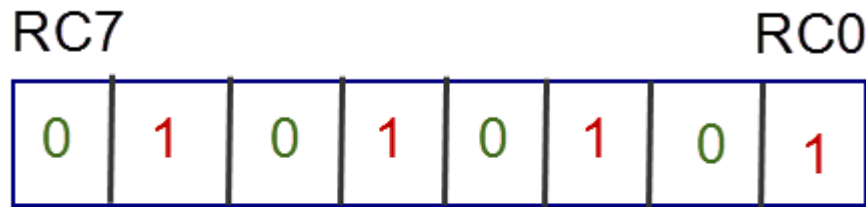


Fig.3
New contents of
LATC 0x55

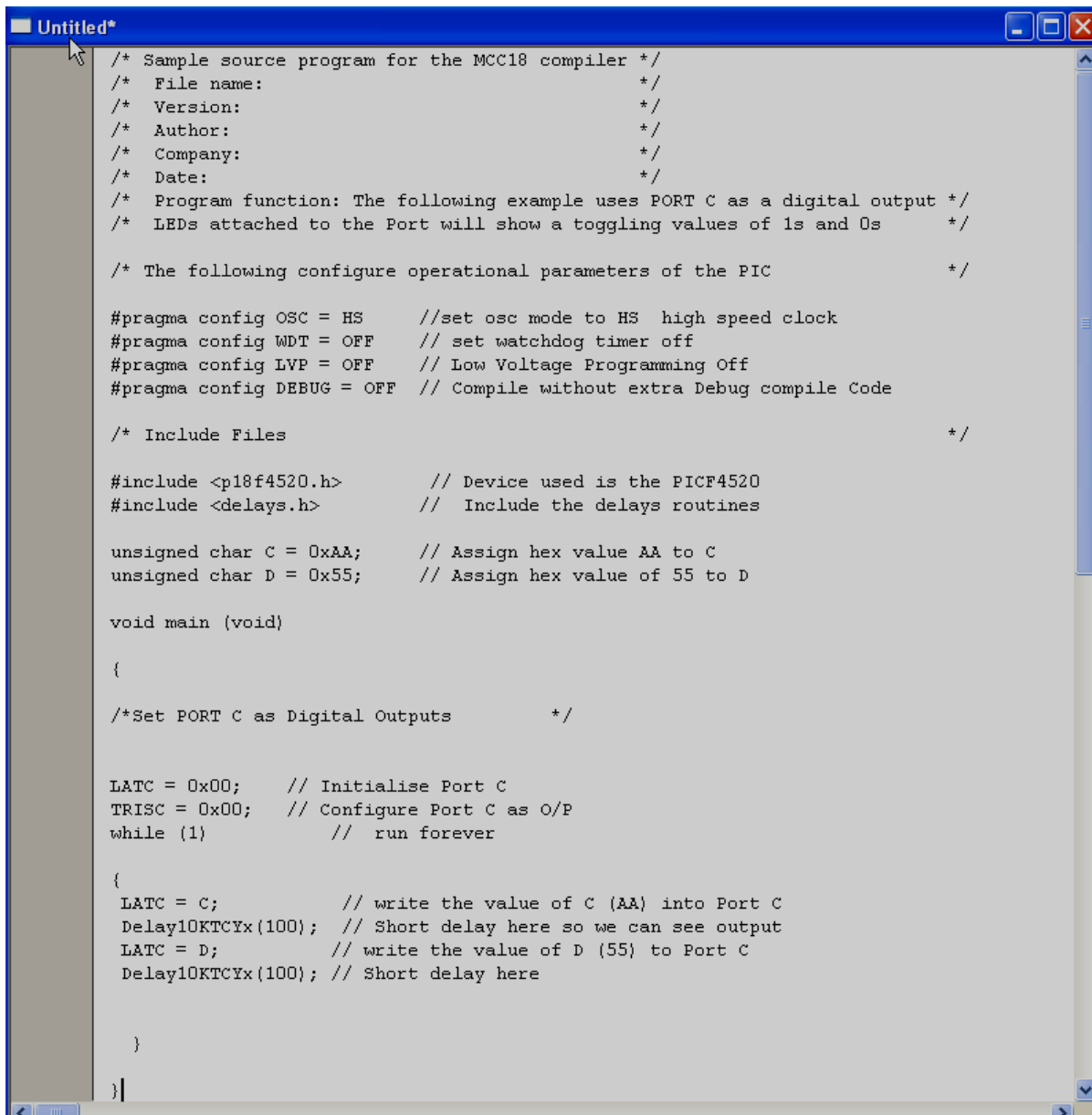
Hex 55

Another delay follows **Delay10KTCYx(100);** which holds the Port C for specified delayed time by calling again the delay function.

Program repeats toggling the values of hex AA and 55 in an endless loop (while(1)) .

Procedure

We will now create an MPLAB C based project, compile the file and test its functionality on the MPALB simulator. We will follow similar procedures to the ones discussed during our MPLAB IDE Assembler laboratory.



```
/* Sample source program for the MCC18 compiler */
/* File name: */
/* Version: */
/* Author: */
/* Company: */
/* Date: */
/* Program function: The following example uses PORT C as a digital output */
/* LEDs attached to the Port will show a toggling values of 1s and 0s */

/* The following configure operational parameters of the PIC */

#pragma config OSC = HS      //set osc mode to HS  high speed clock
#pragma config WDT = OFF     // set watchdog timer off
#pragma config LVP = OFF     // Low Voltage Programming Off
#pragma config DEBUG = OFF   // Compile without extra Debug compile Code

/* Include Files */

#include <p18f4520.h>         // Device used is the PICF4520
#include <delays.h>          // Include the delays routines

unsigned char C = 0xAA;      // Assign hex value AA to C
unsigned char D = 0x55;      // Assign hex value of 55 to D

void main (void)
{
    /*Set PORT C as Digital Outputs */

    LATC = 0x00;             // Initialise Port C
    TRISC = 0x00;            // Configure Port C as O/P
    while (1)                // run forever
    {
        LATC = C;            // write the value of C (AA) into Port C
        Delay10KTCYx(100);   // Short delay here so we can see output
        LATC = D;            // write the value of D (55) to Port C
        Delay10KTCYx(100);   // Short delay here

    }
}
```

Fig. 11 C source file of our project

Experimentation

Exercise 1.

Modify your original source file so that you output incrementing (+1) values to your Port C. Select a longer time delay this time. Provide a Logic Analyser display of your port values (see previous lab Introduction to MPLAB IDE).

Exercise 2

Write a program that starts with the value 0x80 in Port C and right shifts this value from MSB to LSB (i.e. Left to Right) repeating the process in a continuous loop.

Exercise 3

Write a program that sends the following hex equivalent values for ASCII characters "PIC18F" in sequence, one at a time starting with the character "P".

You can either use the ASCII the chart conversion table or another simpler method.

Exercise 4

From our original example make changes to your source file so that instead of using the hex values of 0x55 and 0xAA to toggle the outputs of Port C we only use one of these values and we toggle the output by using a logic operation in C.

Exercise 5

Devise a different way of implementing a delay other than calling the Delay10KTCYx() function.

For help in completing the exercises above please refer to page 23 of the Appendix.

APPENDIX

Keywords

The ANSI C standard defines 32 keywords for use in the C language. The following table shows the ANSI C and the MPLAB C1X keywords, where MPLAB C1X keywords are shown in bold.

_asm	extern	short
_endasm	far	signed
auto	float*	sizeof
break	for	static
case	goto	struct
char	if	switch
const	int	typedef
continue	long	union
default	near	unsigned
do	ram	void
double	register**	volatile
else	return	while
enum	rom	

** has no effect in MPLAB C1X.

Arithmetic and Logical Operations in C

Arithmetic Operations

OPERATION	OPERATOR	DESCRIPTION	SYNTAX
Basic assignment	=	Assign value	a = b
Increment	++	Add 1 to integer	Suffix a++ Prefix ++a
Decrement	--	Subtract 1 from integer	Suffix a-- Prefix --a
Add	+	Addition Integer or float	a + b
Subtract	-	Subtraction Integer or float	a - b
Multiply	*	Multiplication Integer or float	a * b
Divide	/	Division Integer or float	a / b
Divide	/	Division Integer or float	a / b

Comparison/Relational Operations

OPERATION	OPERATOR	DESCRIPTION	SYNTAX
Equal to	==		a == b
Not equal to	!=	Compare two values and	a != b
Greater than	>	return a True or	a > b
Less than	<	False result	a < b
Greater than or equal to	>=	based on the comparison	a >= b
Less than or equal to	<=		a <= b

Bitwise operations

OPERATION	OPERATOR	DESCRIPTION	SYNTAX
AND	&	Integer bitwise Logical AND	a & b
OR		Integer bitwise Logical OR	a b
Exclusive OR	^	Integer bitwise Exclusive OR	a ^ b
NOT	~	Integer bitwise inversion	~a
Bitwise left shift	<<	Integer logical shift left	a << b
Bitwise shift right	>>	Integer logical shift right	a >> b

Logical Operations

OPERATION	OPERATOR	DESCRIPTION	SYNTAX
Logical negation (NOT)	!	Integer logical negation	!a
Logical AND	&&	Logical integer AND operation	a && b
Logical OR		Integer logical OR operation	a b

MPLAB C18 Quick Reference

MPLAB C18 Command Switches

Command	Description									
-?, -h	Display help screen									
-v	Display compiler version number									
-d<macro>[=<text>]	Define a macro									
-fo=<name>	Set object file name									
-fe=<name>	Set error file name									
-i<path>	Add include path									
-k	Default <code>char</code> is unsigned									
-ls	Multi-bank stack									
-nw<n>	Suppress message n									
-O	Equivalent to: -Ob+, -Om+, -On2, -Os+, -Ot+, -Ou+, and -Ow+									
-Oa[+ -]	Default data in access memory									
-Ob[+ -]	Branch optimization									
-Oi[+ -]	Promote to integers									
-Ol[+ -]	Default static locals									
-Om[+ -]	Duplicate string merging									
-On{0 1 2}	Set banking optimizer level									
-Os[+ -]	Code straightening									
-Ot[+ -]	Tail merging									
-Ou[+ -]	Unreachable code removal									
-Ow[+ -]	WREG content tracking									
-p=<processor>	Set processor									
-q	Quiet mode									
-w{1 2 3}	Set warning level									
-m{s l}	Select memory model <div><table><tr><th colspan="2"></th><th>ROM</th></tr><tr><td>s</td><td>small</td><td>near</td></tr><tr><td>l</td><td>large</td><td>far</td></tr></table></div>			ROM	s	small	near	l	large	far
		ROM								
s	small	near								
l	large	far								

MPLAB C18 Libraries and Precompiled Object Files

File	Use
clib.lib	Standard C routines, math routines, startup code
c018i.o	Startup code with initialized data support
c018.o	Startup code without initialized data support
p18?????.lib	Peripheral library routines and SFR definitions

???? = processor type (e.g., C452 for PIC18C452, F020 for PIC18F020.)

MPLAB C18 Types

Type	Bit Width	Range
void	N/A	none
char	8	-128 to 127
unsigned char	8	0 to 255
int	16	-32,768 to 32,767
unsigned int	16	0 to 65,535
short	16	-32,768 to 32,767
unsigned short	16	0 to 65,535
short long	24	-8,388,608 to 8,388,607
unsigned short long	24	0 to 16,777,215
long	32	-2,147,483,648 to 2,147,483,647
unsigned long	32	0 to 4,294,967,295
float	32	1.7549435E-38 to 6.80564693E+38
double	32	1.7549435E-38 to 6.80564693E+38

Common MPLAB C18 Type Modifiers

Modifier	Use
const	Variable will not be modified
far	Variable is paged/banked regardless of memory model selected
extern	Variable is allocated in another module
near	Variable is not paged/banked regardless of memory model selected
ram	Locate object in data memory
rom	Locate object in program memory
static	Variable is retained unchanged between executions of the defining block.
volatile	Variable may change from other sources (e.g., input port)

MPLAB C18 Interrupts

To create an interrupt service routine in your MPLAB C18 code, no additional libraries need be included. Simply do the following:

- Create a code section at the interrupt vector that contains a `goto isr` statement, either using inline assembly or a separate assembly file.
- Declare your interrupt routine in your source code using one of the following statements:

High-priority interrupts – W, BSR, and STATUS are saved in shadow registers

```
#pragma interrupt <isr> [save=symbol-list]
```

Low-priority interrupts – W, BSR, and STATUS are saved on the software stack

```
#pragma interruptlow <isr> [save=symbol-list]
```

The following registers are managed by the compiler and should not be saved using `save=symbol-list`.

FSR0*	PCL
FSR1	STATUS
FSR2	WREG
	BSR

* The compiler will automatically save FSR0 if needed.

If your ISR calls non-ISR functions, the temporary data section must be saved. this is done using the section qualifier on the `save=` keyword.

```
#pragma interruptlow <isr> [save=section(".tmpdata")]
```

MPLAB C18 Inline Assembly

MPLAB C18 has an internal assembler with a syntax similar to the MPASM assembler, except that comments must be in the C (`/* */`) or C++ (`//`) style. The block of assembly code must begin with `_asm` and end with `_endasm`. For example:

```
_asm
movlw 7      // Load 7 into WREG
movwf PORTB  // and send it to PORTB
_endasm
```



Document URL ScanCode