Contents

Introduction

Flash memory for program

Used to store the program. This is where the C source code that we write ends up. Once converted into PIC machine code our program is stored in this non-volatile memory area. This memory is 16 bits wide since each instruction is 16 bits wide. (There are 4 instructions that take two locations means they will return all '0's (a NOP instruction). all others take 1 location).

The PIC18F46K22 implements program memory size of 32,768 words (2^15) or 65,536 (2^16) bytes. Note, only even addresses apply since nearly all instructions take 2 bytes(16 bits). Therefore the least significant address bit is fixed at 0 and addresses are therefore always even.

The RESET vector is at 0000h

The high priority interrupt vector is at 0008h.

The low priority interrupt vector is at 0018h.

SRAM for SFRs and GFRs

SRAM – maximum of 256 bytes per bank due to fact there is only 8 bits allocated in the instruction format to address it.

This is where the variables are stored when the program is running. The PIC documentation calls them *General Purpose Registers* in that you can use them to hold whatever values the program is working on.

When a variable is declared in 'C' it is allocated some of the GPRs Part of the RAM is used for SFRs (Special Function Registers) used to configure the hardware. For the PIC18 the SFRs are in the last bank of RAM.

Special Function Registers(SFRs)

Memory locations that control the functionality of the microcontroller. Specifically the values written to the SFRs determine how the core hardware (example clocking , interrupts) and peripherals(example, Timers, ADCs, UART) operate.

General Purpose Registers(GPRs)

Memory locations used to store general data, for example when a variable is declared in 'C', the compiler allocates some GPRs to the variable.

EEPROM

EEPROM – maximum of 1024 bytes used to hold non-volatile data.

Instruction Set Summary

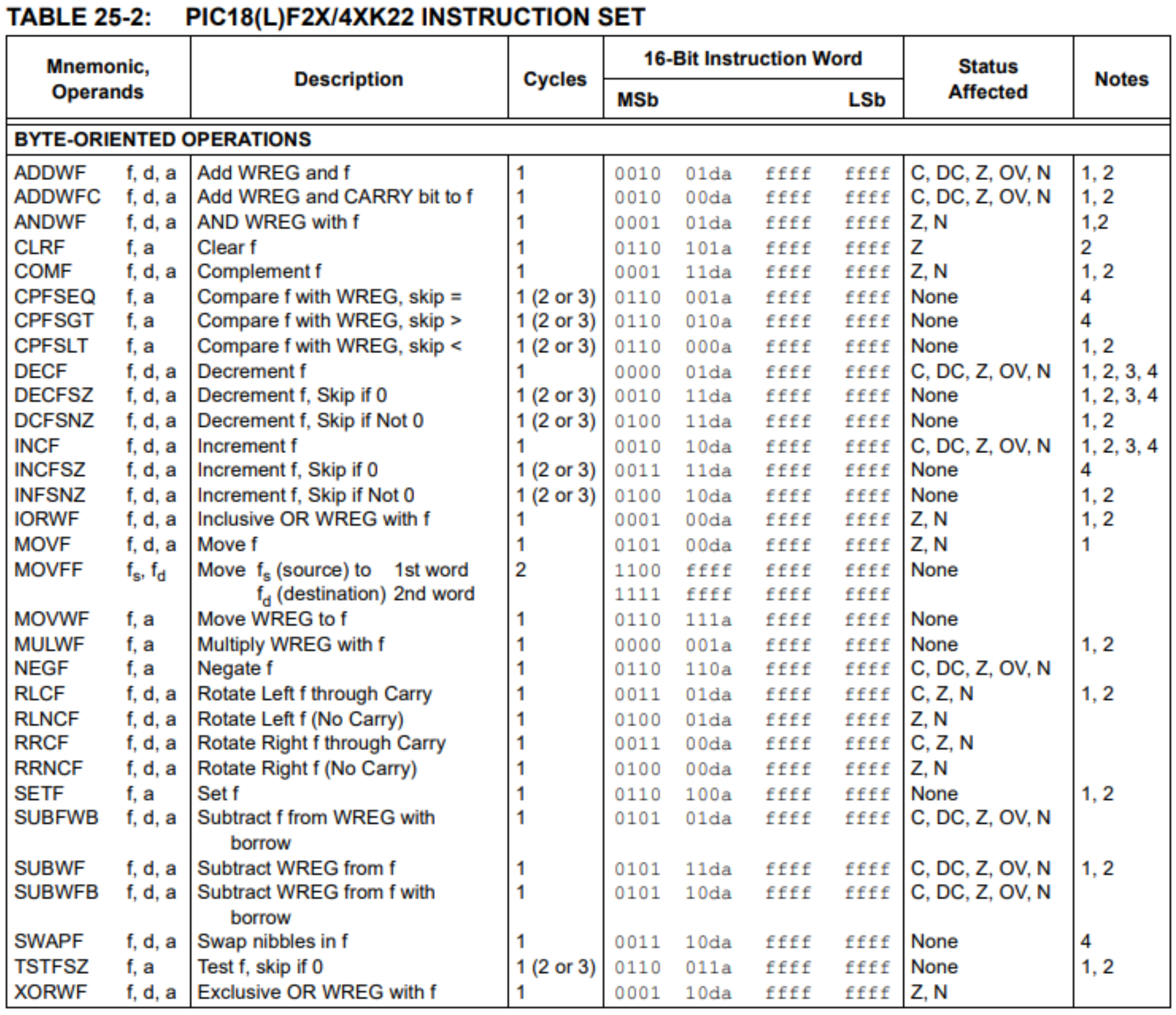


TABLE 25-2: PIC18(L)F2X/4XK22 INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word MSb			LSb	Status Affected	Notes
BYTE-ORIENTED OPERATIONS									
ADDWF	f, d, a	Add WREG and f	1	0010	01da	ffff	ffff	C, DC, Z, OV, N	1, 2
ADDWFC	f, d, a	Add WREG and CARRY bit to f	1	0010	00da	ffff	ffff	C, DC, Z, OV, N	1, 2
ANDWF	f, d, a	AND WREG with f	1	0001	01da	ffff	ffff	Z, N	1,2
CLRF	f, a	Clear f	1	0110	101a	ffff	ffff	Z	2
COMF	f, d, a	Complement f	1	0001	11da	ffff	ffff	Z, N	1, 2
CPFSEQ	f, a	Compare f with WREG, skip =	1 (2 or 3)	0110	001a	ffff	ffff	None	4
CPFSGT	f, a	Compare f with WREG, skip >	1 (2 or 3)	0110	010a	ffff	ffff	None	4
CPFSLT	f, a	Compare f with WREG, skip <	1 (2 or 3)	0110	000a	ffff	ffff	None	1, 2
DECF	f, d, a	Decrement f	1	0000	01da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4
DECFSZ	f, d, a	Decrement f, Skip if 0	1 (2 or 3)	0010	11da	ffff	ffff	None	1, 2, 3, 4
DCFSNZ	f, d, a	Decrement f, Skip if Not 0	1 (2 or 3)	0100	11da	ffff	ffff	None	1, 2
INCF	f, d, a	Increment f	1	0010	10da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4
INCFSZ	f, d, a	Increment f, Skip if 0	1 (2 or 3)	0011	11da	ffff	ffff	None	4
INFSNZ	f, d, a	Increment f, Skip if Not 0	1 (2 or 3)	0100	10da	ffff	ffff	None	1, 2
IORWF	f, d, a	Inclusive OR WREG with f	1	0001	00da	ffff	ffff	Z, N	1, 2
MOVF	f, d, a	Move f	1	0101	00da	ffff	ffff	Z, N	1
MOVFF	f_s, f_d	Move f_s (source) to 1st word	2	1100	ffff	ffff	ffff	None	
		f_d (destination) 2nd word		1111	ffff	ffff	ffff		
MOVWF	f, a	Move WREG to f	1	0110	111a	ffff	ffff	None	
MULWF	f, a	Multiply WREG with f	1	0000	001a	ffff	ffff	None	1, 2
NEGF	f, a	Negate f	1	0110	110a	ffff	ffff	C, DC, Z, OV, N	
RLCF	f, d, a	Rotate Left f through Carry	1	0011	01da	ffff	ffff	C, Z, N	1, 2
RLNCF	f, d, a	Rotate Left f (No Carry)	1	0100	01da	ffff	ffff	Z, N	
RRCF	f, d, a	Rotate Right f through Carry	1	0011	00da	ffff	ffff	C, Z, N	
RRNCF	f, d, a	Rotate Right f (No Carry)	1	0100	00da	ffff	ffff	Z, N	
SETF	f, a	Set f	1	0110	100a	ffff	ffff	None	1, 2
SUBFWB	f, d, a	Subtract f from WREG with borrow	1	0101	01da	ffff	ffff	C, DC, Z, OV, N	
SUBWF	f, d, a	Subtract WREG from f	1	0101	11da	ffff	ffff	C, DC, Z, OV, N	1, 2
SUBWFB	f, d, a	Subtract WREG from f with borrow	1	0101	10da	ffff	ffff	C, DC, Z, OV, N	
SWAPF	f, d, a	Swap nibbles in f	1	0011	10da	ffff	ffff	None	4
TSTFSZ	f, a	Test f, skip if 0	1 (2 or 3)	0110	011a	ffff	ffff	None	1, 2
XORWF	f, d, a	Exclusive OR WREG with f	1	0001	10da	ffff	ffff	Z, N	

Figure1: Instruction Set Summary

Exercise 2

In Exercise 2 we look at very simple C files and investigate the assembly code and machine code generated by the compiler.

We can look at C code translate the assembly instruction and machine code by the compiler.

The nature/architecture of the microcontroller is best determined by looking at the assembly instructions available in addition to the block diagram. For example if there is a hardware multiplier as part of the CPU; Then there is a MUL instruction.

We click the launch the debugg, click the step into and gain a better understanding of microcontroller operation. We can look at Address, Assembly Instructions and Machine Code in the program memory. There is a direct one to one translation between assembly and machine code. Machine code is generally represented in hex for easy reading.

Part 1: Investigating Program Memory and Assembly

Main Code

```c
#include "mcc_generated_files/mcc.h"
void Initial(void);
void delay_1s(void);
void main(void)
{
    unsigned char count = 0;
    // Initialize the device
    SYSTEM_Initialize();
    Initial();
    while (1)
    {
        // Add your application code
        count ++;
        LATB = count;
        delay_1s();
        LATB = 0xAA;
        delay_1s();
    }
}



void Initial(void)
{
    LATB = 0xFF;
    delay_1s();
    delay_1s();
    LATB = 0x00;
}

void delay_1s(void)
{
    unsigned char i;
    for (i=0;i<25;i++)
            __delay_ms(40);  //max value is 40 since this depends on the _delay() function which has a max number of cycles

}

```

Answer the Questions

A: Instruction at the reset location in program memory is MOVLB 0x0.

A: GOTO 0xFF74 is happening before the processor reaches 'main'.

A: State the program memory' address where the 'main' function starts to FF74.

A: State the program memory' address where the 'initial' function starts to FF64

Q: Determine how many locations in program memory the 'initial' function uses.

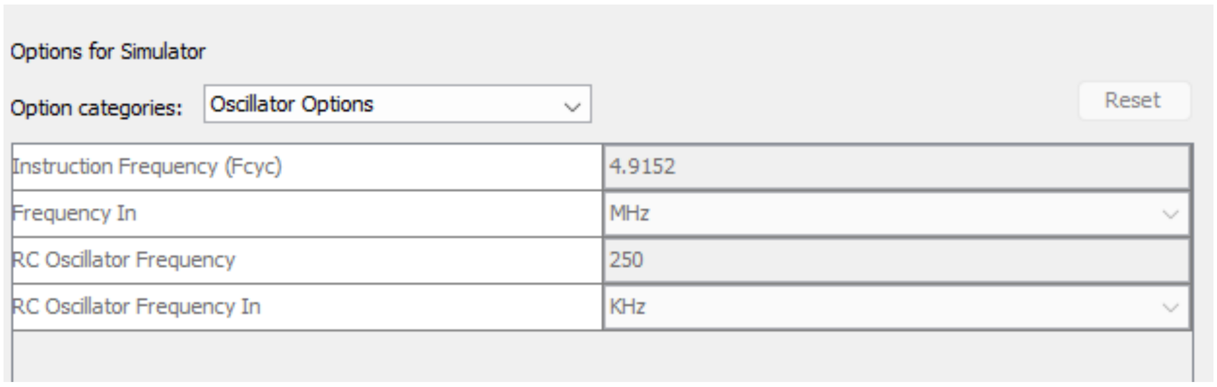
A:

1- SETF LATB, ACCESS
2- CALL 0xFF9A, 0
3- NOP
4- CALL 0xFF9A, 0
5- NOP
6- MOVLW 0x0
7- MOVWF LATB, ACCESS

8- RETURN 0

8 locations. every location is 2 bytes

Q: Use the simulator to determine the real time it takes for the initial function to run. Make sure the instruction frequency is set correctly for the simulator.

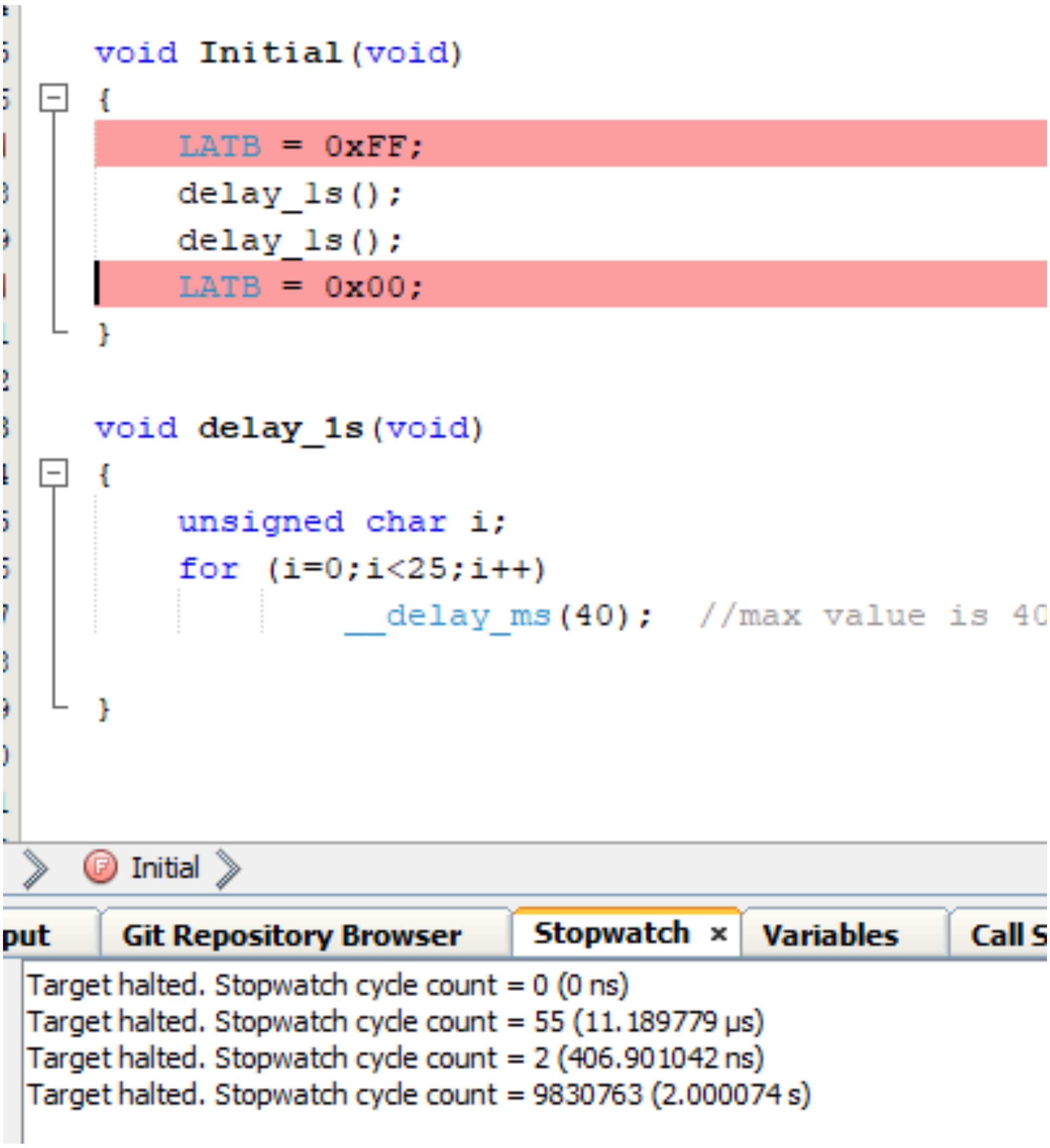


Options for Simulator

Option categories: Oscillator Options

Instruction Frequency (Fcyc)	4.9152
Frequency In	MHz
RC Oscillator Frequency	250
RC Oscillator Frequency In	KHz

Figure2: Instruction Frequency



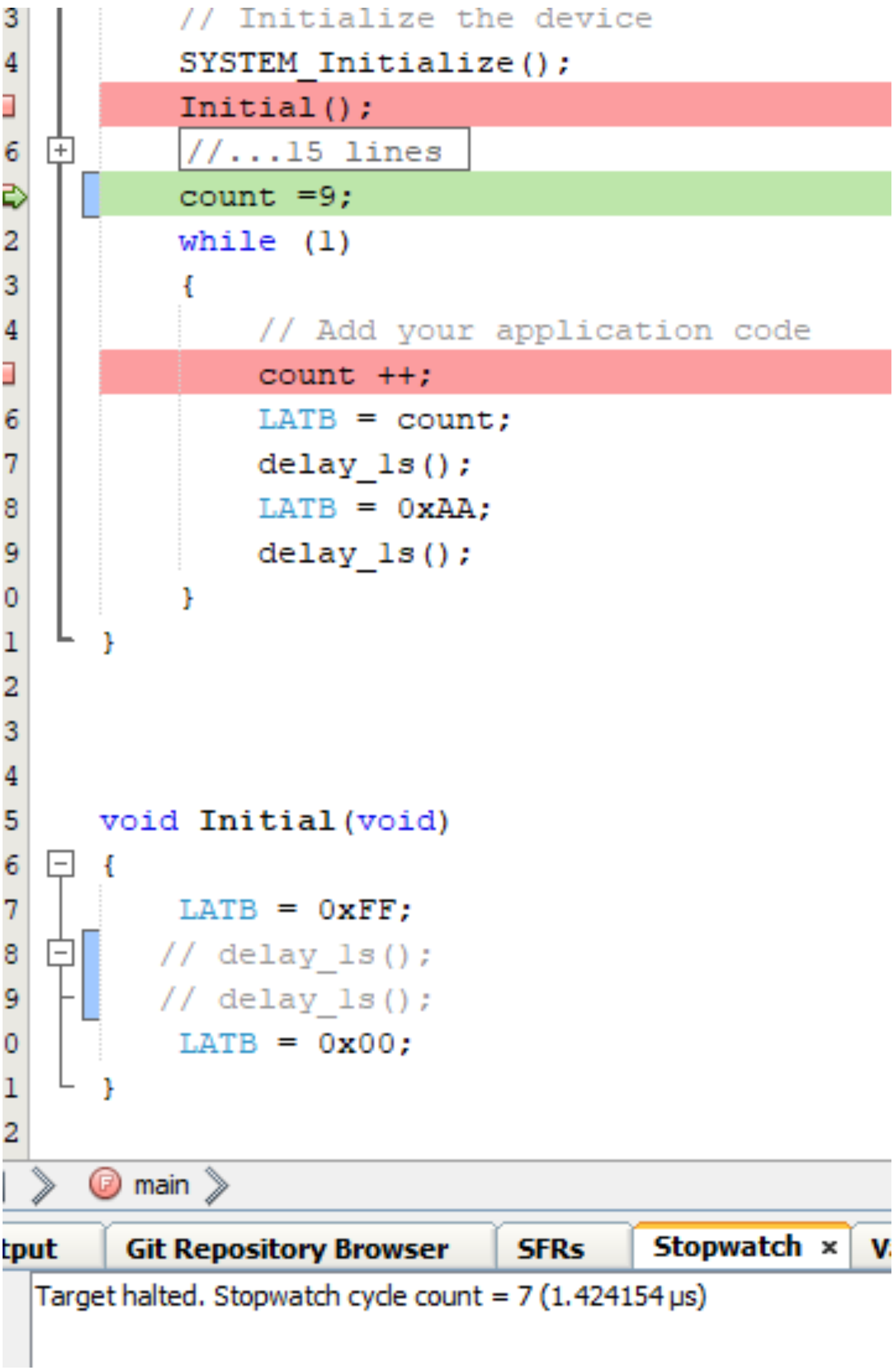
```
void Initial(void)
{
    LATB = 0xFF;
    delay_1s();
    delay_1s();
    LATB = 0x00;
}

void delay_1s(void)
{
    unsigned char i;
    for (i=0;i<25;i++)
            __delay_ms(40);  //max value is 40
}
```

```
Target halted. Stopwatch cycle count = 0 (0 ns)
Target halted. Stopwatch cycle count = 55 (11.189779 µs)
Target halted. Stopwatch cycle count = 2 (406.901042 ns)
Target halted. Stopwatch cycle count = 9830763 (2.000074 s)
```

Figure2: Initial Function Time

A: About 2 s

Q: Comment out any delay functions. Build and look at the assembly instructions for this function. Calculate how long you expect it to take to execute the function. Use the simulator to verify your calculation.



```
    // Initialize the device
    SYSTEM_Initialize();
    Initial();
    //...15 lines
    count =9;
    while (1)
    {
        // Add your application code
        count ++;
        LATB = count;
        delay_1s();
        LATB = 0xAA;
        delay_1s();
    }
}



void Initial(void)
{
    LATB = 0xFF;
   // delay_1s();
   // delay_1s();
    LATB = 0x00;
}
```

```
Target halted. Stopwatch cycle count = 7 (1.424154 µs)
```

Figure3: Delay Function Time

A:

An instruction generally takes 1 instruction cycle to execute though some take 2 and a few 3 instruction cycles.

The instruction frequency is ¼ of the system clock frequency.

Therefore instruction period is 4 * system clock period.

There are total 7 instruction

19.6608/4= 4.9152

1/4.9152 =0.2

Instruction 0.2*7=1.4 us

Q: State the address in SRAM of the LATB Special Function Register (SFR).

A: 0 x F8A;

Q: State the address in SRAM of the ‘count’ variable.

A: 0 x 3

Q: Determine how the infinite while loop is implemented in assembly. Paste the assembly code for this in your report.

```c
unsigned char countc __at (0x345);   //can only apply addresses to global variables

void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();
    Initial();

    countc = 27;

    while (1)
    {

        LATB = countc;
        countc++;


    }
}
```

A:

INCF 0x45, F, BANKED

GOTO 0xFF8C

GOTO 0x0

Q: Make note of the assembly which implements the following line of C

‘TRISB = 0x00;’

```
void Initial(void)
{
    TRISB = 0x00;
    LATB = 0xFF;
    delay_1s();
    delay_1s();
    LATB = 0x00;
}

```

A:

MOVLW 0x0

MOVWF TRISB, ACCESS

Q: Also make note of the assembly and machine code generated for the C statement ‘LATB=0x01’;

```
void Initial(void)
{

    LATB = 0x01;
    delay_1s();
    delay_1s();
    LATB = 0x00;
}

```

A:

MOVLW 0x1

MOVWF LATB, ACCESS

Part 2: Looking at Access RAM

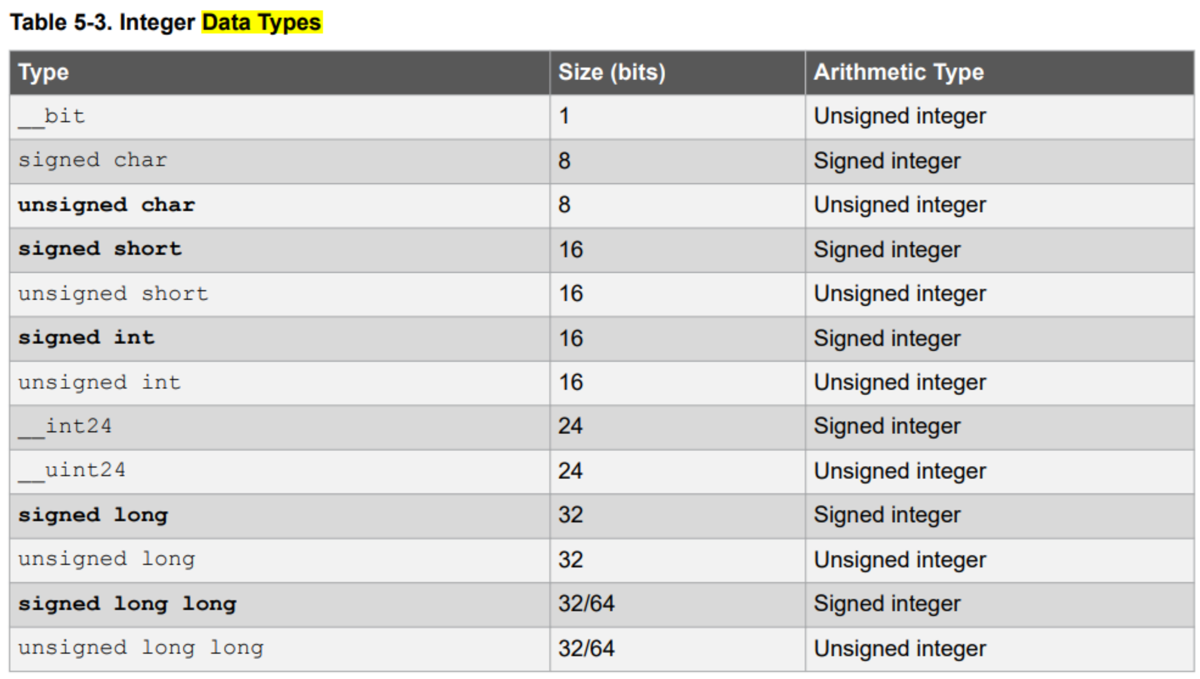


Table 5-3. Integer Data Types

Type	Size (bits)	Arithmetic Type
`__bit`	1	Unsigned integer
`signed char`	8	Signed integer
`unsigned char`	8	Unsigned integer
`signed short`	16	Signed integer
`unsigned short`	16	Unsigned integer
`signed int`	16	Signed integer
`unsigned int`	16	Unsigned integer
`__int24`	24	Signed integer
`__uint24`	24	Unsigned integer
`signed long`	32	Signed integer
`unsigned long`	32	Unsigned integer
`signed long long`	32/64	Signed integer
`unsigned long long`	32/64	Unsigned integer

Figure4: Data Type

A:The range of GPR Address is 000h- FFFh, the range of SFR Address is F60h- FFFh

Q: Explain the difference in terms of SRAM memory space, program memory space and execution time between the following two variable initializations which happen in the main function. Assuming they are implemented.

unsigned char count = 0;

unsigned long long_count = 0;

A:

SRAM Memory Space:

unsigned char count = 0; This declaration allocates a small amount of SRAM memory to store the count variable. An unsigned char typically occupies one byte (8 bits) in SRAM.

unsigned long long_count = 0; In contrast, unsigned long is usually a larger data type, and it requires more SRAM space. The exact size of an unsigned long can vary based on the compiler and the target architecture, but it typically occupies at least 4 bytes (32 bits) or more in SRAM.

The impact on program memory space and execution time is generally minor, and it mostly depends on the specific use of these variables in the program's logic and any subsequent operations performed on them.

Q: If 'count' is a variable declared as unsigned char and the compiler decides it goes at address 0x007 in SRAM, generate the assembly code and machine code you expect for the following C statement. Pay particular attention to the access field bit in the machine code.

count = 9;

A:

Assembly code: MOVLW 0x9

Machine code: 0E09

In MPLAB® XC8 C Compiler User's Guide for PIC® MCU

Any object which has static storage duration and which has file scope can be placed at an absolute address in data memory, thus all but static objects defined inside a function and stack-based objects can be made absolute.

For example:

volatile unsigned char Portvar _at(0x06);

will declare a variable called Portvar located at 06h in the data memory. Note that the _at () construct can be placed before or after the variable identifier in the definition, but to be compatible with the C90 standard, it should be place after the identifier.

Note: Defining absolute objects can fragment memory and can make it impossible for the linker to position other objects. If absolute objects must be defined, try to place them at either end of a memory bank so that the remaining free memory is not fragmented into smaller chunks.

Q: Explain, for this compiler, how to place a variable at a specific location or absolute address in RAM.

A：

So _at (Address)

The range of Access RAM Low address is 00h – 5Fh.

For example:

unsigned char countc __at (0x00h)

Q: If a variable called 'countc' has an absolute address of 0x345, write the assembly instructions that you expect to be generated for the following line of C code.

countc = 27;

```c
unsigned char countc __at (0x345);   //can only apply addresses to global variables
void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();
    Initial();
    countc = 27;
    while (1)
    {
        LATB = countc;
        countc++;
    }
}

```

A:

MOVLW 0x1B

MOVLB 0x3

MOVWF 0x45, BANKED

Part3： Dynamic Memory Allocation

Create a table with three columns. In the first column put the variable name and the function it is declared in. In the second column put the memory address the variable uses. In the third column say whether the variable is dynamic or static.

Variable Name	Function	Memory Address	Dynamic/Static
unsigned char countc_at(0x261)	globale variables		Static
unsigned char main_i=0	main()	0x4	Dynamic
unsigned char i	initial()	0x3	Dynamic

unsigned char i	delay()	0x2	Dynamic
unsigned char j	Flash_fast()	0x3	Dynamic
unsigned char j	Flash_fast2()	0x2	Dynamic

Char 1: Dynamic Variable and Static Variable

Part 4: An array in RAM

Write some code using a ‘for’ loop to initialize all the elements of the array to 0. Place this inside main but before the infinite loop.

```c
void main() {
    // Define an array
    int myArray[10]; // Change the size according to your needs

    // Initialize all elements of the array to 0
    for (int i = 0; i < 10; i++) { // Change the size accordingly
        myArray[i] = 0;
    }

    // Infinite loop
    while (1) {
        // Your infinite loop code here
    }

}
```

In this code, an array myArray of size 10 is declared and then initialized with all elements set to 0 using a 'for' loop.

Q: Notice if indirect addressing assembly commands are used when accessing the array. Look at section 5.6.3 in the datasheet. List the commands used.

A:

Indirect addressing assembly commands are used when accessing the array.

Indirect File Operands (INDFs) that permit automatic manipulation of the pointer value with auto-incrementing, auto-decrementing or offsetting with another value. This allows for efficient code, using loops, such as the example of clearing an entire RAM bank

```
         LFSR    FSR0, 100h ;
NEXT     CLRF    POSTINC0    ; Clear INDF
                             ; register then
                             ; inc pointer
         BTFSS   FSR0H, 1    ; All done with
                             ; Bank1?
         BRA     NEXT        ; NO, clear next
CONTINUE                     ; YES, continue
```

Figure5： clearing an entire RAM bank

LFSR FSR0, 100H; // Move 100H to FSR0

CLRF POSTINCO; // CLRF: This part clears the content of the INDF register, setting it to zero.

POSTINC0: After clearing INDF, the pointer is incremented by one to point to the next memory location.

POSTINC: accesses the location to which the FSR points, then automatically increments the FSR by 1 afterwards

This type of operation is commonly used in assembly programming when working with arrays.

BTFSS FSR0H, 1// if the first bit (bit 1) in the high byte of the FSR0 register is set to 1, the instruction will skip to the label or address specified after it. If this bit is clear (0), it won't perform the skip and will continue executing the next instruction.

BRA NEXT // instructs the program to unconditionally jump to the location specified by the "NEXT" label or address, without considering any conditions or flags.

Q: How many bytes of RAM does the array use and what is the address range when

SIZE = 10.

A:

An unsigned char typically represents 1 byte of data. In memory, it's the smallest addressable unit, and it can hold values in the range of 0 to 255. If SIZE = 10,

RAM Usage = Size of each element * Number of elements

RAM Usage = 1 bytes/element * 10 elements

RAM Usage = 10 bytes

Address rang is 00h-0Ah

Q: How many bytes of RAM does the array use and what is the address range when

SIZE = 256.

A:

RAM Usage = Size of each element * Number of elements

RAM Usage = 1 bytes/element * 256 elements

RAM Usage = 256 bytes

Address rang is 00h-FFh

Q: How many bytes of RAM does the array use and what is the address range when

SIZE = 260.

A:

RAM Usage = Size of each element * Number of elements

RAM Usage = 1 bytes/element * 260 elements

RAM Usage = 260 bytes

Address rang is 00h-103h

Q: How many bytes of RAM does the array use and what is the address range when

SIZE = 10 but the data type is changed to unsigned int.

A:

unsigned int arry[10];

An unsigned int typically represents 4 byte of data. In memory, it's the smallest addressable unit, and it can hold values in the range of 0 to 65536. If SIZE = 10,

RAM Usage = Size of each element * Number of elements

RAM Usage = 4 bytes/element * 10 elements

RAM Usage = 40 bytes

Address rang is 00h-28h

Optional Part 5:

In MPLAB® XC8 C Compiler User's Guide for PIC® MCU

5.4.4.2 Absolute Objects In Program Memory

Any const-qualified object which has static storage duration and which has file scope can be placed at an absolute address in program memory

For example:

```
   const int settings[]__at(0x200) = { 1, 5, 10, 50, 100 };
```

will place the array settingsat address 0x200 in the program memory.

Note that the___at() construct can be placed before or after the variable identifier in the definition, but to becompatible with the C90 standard, it should be place after the identifier.

An uninitialized extern const object can be made absolute and is useful when you want to define a placeholderobject that does not make a contribution to the output file.

Q: Placing a string or Lookup Table (Read only) in Program Memory.

A:

```
        const char myString[] _at(0xA7h) = "Good Morning!";

        const int lookupTable[] _at(0xA7h) = {1, 2, 3, 4, 5};
```

Briefly research how to place data and strings in the FLASH Memory for read only and what assembly commands are used to access this information.

Conclusion

Through this exercise, we look at very simple C files and investigate the assembly code and machine code generated by the compiler.

We can look at C code translate the assembly instruction and machine code by the compiler.

We click the launch the debugg, click the step into and gain a better understanding of microcontroller operation. We can look at Address, Assembly Instructions and Machine Code in the program memory. There is a direct one to one translation between assembly and machine code. Machine code is generally represented in hex for easy reading. However there is not a direct translation between 'C' code and assembly instructions.

We know that the File Memory, SFRs, GPRs, Access RAM Address rang, some assembly instruction function by datasheet.