Introduction

This project’s purpose is to become familiar with analog to digital conversion on a PIC micro-controller. Four programs demonstrate: assembly timing, A/D conversions, A/D conversion rates, and RC circuit time constants.

The micro-controller is a 28 pin DIP PIC16F876 clocked at 20 MHz manufactured by Microchip. The PIC voltage input is +5 volts DC via a $\mu$A7805 dc/dc voltage regulator. The compiler is the CCS C compiler (v. 3.207) for 14 bit PIC chips. C compilation occurs on a x86 based PC. Data transfer between the PIC and the PC is through a 9 pin serial cable at 19200 baud. All programming and experiments were performed by Charles O’Neill.

1 CCS Compiler Assembly

1.1 Type Declarations

This part investigates how variables are declared and stored in assembly as compiled from the C source. The code is given in asm.c in the Code Listings section (p.14). The first test is storing the value 13 to an integer.

................. /* Type Declaration Experiment */
................. int       a = 13;
0231: MOVWL 0D
0232: MOVWF 2A

Line 0231 moves the literal 0x0D (13 decimal) to the W register. Line 0232 copies the W register to file 0x2A. The assembler operates in hexadecimal for both addresses and literals.

Next, an integer pointer is defined to the integer a. Remembering that the above declaration stores ‘A’ in 0x2A, the pointer address should also be 0x2A stored in the memory address 0x2B.
Next, a signed integer is declared. The compiler recalls that 0x0D (13 decimal) is already in the W register, so this declaration requires only a copy from the W register to the address of b, which is 0x2C.

.................... signed int b = 13;

Next, a signed integer is defined as -20 decimal. The 16F876’s integer is 8 bits for 256 total representations. So, -20 decimal signed maps to 236 decimal unsigned (0xEC).

.................... signed int c = -20;

The signed integer has an 8bit range of −127–128. The next few instructions test the compiler’s assumptions for variable overflows.

.................... signed int d2 = -128;

An overflow of -128 compiles to 0x80, which maps to 128 decimal. 255 decimal maps to 0xFF (255 unsigned decimal or -127 signed decimal). 256 decimal gives a silent compiler failure which stores zero (0x00). No visible or formal compiler errors occurred. Apparently, the compiler maps out-of-range signed integers to the corresponding signed integer.

Characters are represented as integers starting at decimal 65 (0x41) for uppercase and 97 (0x61) for lowercase. The character e is located at a+5 97 + 4 = 101 (0x65).

.................... char e = 'e';

An array of characters is similar. Again, the compiler recognizes that c is already in the W register for f[4]=c.

.................... char f[4] = {'a', 'b', 'c', 'c'};
1.2 Math Operations

This section investigates the CCS compiler’s math operations assembly code. Eight bit operations will be shown; however, 16 bit and floating point math operations will be compared.

1.2.1 8 bit (1 Byte) Math Operations

The first operation is addition. The eight bit addition requires 2 operations: moving a to the register, and adding and moving a to its memory location.

```assembly
.................... /* Addition */
.................... a = a + 1;
00B1: MOVLW 01
00B2: ADDWF 37,F
```
Incrementing a requires 1 assembly operation: INCF. The operation savings of incrementing obviously only occurs for an addition of 1.

```assembly
.................... /* Increment++ */
.................... a++;  
00B3: INCF 37,F
```
Subtraction is similar to addition. The order of operation for subwf is register-file.

```assembly
.................... /* Subtract */
.................... a = a - b;
00B4: MOVF 38,W
00B5: SUBWF 37,F
```

Multiplication is much more complicated and requires a specialized goto to the multiplication routine at 0x063. This can be verified with the symbolic map: 0063 @MUL88. Division is similar to multiplication.

```assembly
.................... /* Multiply */
.................... a = a * b;
00B6: MOVF 37,W
00B7: MOVWF 39
00B8: MOVF 38,W
00B9: MOVWF 3A
00BA: GOTO 063
```
00BB: MOVF 78,W
00BC: MOVWF 37

...................... /* Divide */
...................... a = a / b;
00BD: MOVF 37,W
00BE: MOVWF 39
00BF: MOVF 38,W
00C0: MOVWF 3A
00C1: GOTO 08A
00C2: MOVF 78,W
00C3: MOVWF 37

1.2.2 Multi Byte and Float Math Operations

Comparing the 8 bit math operations to 16 bit operations shows that the PIC
effectively uses two bytes to store a 16 bit integer. All mathematical operations
requires operations on both bytes. Thus, 16 bit arithmetic requires approxi-
mately twice the time as 8 bit arithmetic. For example, the 16 bit increment
has an increment for the lower byte (0x37) and possibly the upper byte if the
lower byte overflows.

...................... /* Increment++ */
...................... a++;  
011D: INCF 37,F
011E: BTFSC 03.2
011F: INCF 38,F

Floating point math operations are significantly more complicated. A single
float addition operation requires 28 operations plus a call. Remembering a
previous project, floating point arithmetic requires approximately 2 order of
magnitude more time.

1.3 Logical Operations and Branching

1.3.1 Equality

The next instruction a==1 — an equality operator — results in code that stores 0
in the W register if a equals 0, or 1 if a does not equal 0. This is valid assembly
code that does nothing. Interestingly, while the C code does nothing, the result
could be useful for creating a stack based processing method.

...................... a=1; // compare only (1)
0259: DECSZ 2A,W
025A: GOTO 25C
025B: GOTO 25D
025C: MOVFW 00
Next, a non-unity integer is tested for equality. The method for logical operations appears to be: subtract and test for zero. For \( a==13 \), the value of \( a \) is moved to the W register; 13 (0x0D) is subtracted from the W register; if the result in the status register bit 2 —zero bit in the status register— does not equal 0 then set the W register to 0.

\[
\begin{align*}
\text{.................} & \quad \text{a==13;} \quad // \text{ compare only (13)} \\
025D: & \quad \text{MOVF} \ 2A,W \\
025E: & \quad \text{SUBLW} \ 0D \\
025F: & \quad \text{BTFSS} \ 03.2 \\
0260: & \quad \text{MOVLW} \ 00
\end{align*}
\]

### 1.3.2 If else

An if-else statement is similar with the addition of bit-test-file-skip-set (BTFSS) assembly statements and GOTO statements. First, a compare occurs (4D1 through 4D3) with a bit test (BTFSS) of the ‘zero’ status register (bit 2). If BTFSS finds an inequality and doesn’t skip an instruction, a goto moves to the next compare at 4D8. Otherwise, a literal is moved first to the W register and then to a memory location.

\[
\begin{align*}
\text{.................} & \quad \text{if(a==14) \{} \quad // \text{ If else} \\
04D1: & \quad \text{MOVF} \ 2A,W \\
04D2: & \quad \text{SUBLW} \ 0E \\
04D3: & \quad \text{BTFSS} \ 03.2 \\
04D4: & \quad \text{GOTO} \ 4D8 \\
\text{.................} & \quad \text{a=2;} \\
04D5: & \quad \text{MOVLW} \ 02 \\
04D6: & \quad \text{MOVWF} \ 2A \\
\text{.................} & \quad \text{\} else if(a==4)} \{ \\
04D7: & \quad \text{GOTO} \ 4DE \\
04D8: & \quad \text{MOVF} \ 2A,W \\
04D9: & \quad \text{SUBLW} \ 04 \\
04DA: & \quad \text{BTFSC} \ 03.2 \\
\text{.................} & \quad \text{a=a;} \\
\text{.................} & \quad \text{\} else {} \\
04DB: & \quad \text{GOTO} \ 4DE \\
\text{.................} & \quad \text{a=3;} \\
04DC: & \quad \text{MOVLW} \ 03 \\
04DD: & \quad \text{MOVWF} \ 2A \\
\text{.................} & \quad \text{\} } \\
\text{.................} & \quad \text{\}
\end{align*}
\]

### 1.3.3 Conditional Operator

The conditional operator — \(( \ ) \ ? : \ —\) is similar to the if statement but requires significantly fewer operations, 8 vs 14, for 2 branch operations. The condi-
tional operator uses bit-test (BTFSS) and goto operations. The series of goto statements deflect the fall-out to the memory write (MOVWF).

```assembly
0272: MOVF 2C,W
0273: SUBLW 0D
0274: BTFSS 03.2
0275: GOTO 278
0276: MOVWF 02
0277: GOTO 279
0278: MOVLW 03
0279: MOVWF 2A
```

### 1.3.4 GOTO

The infamous goto is not generally accepted as a good programming operator. The C to assembly compilation is simple: the C label is hard coded to an address. The CCS compiler did not catch the dead `a++;` code.

```assembly
04E6: GOTO 4E8
```

### 1.3.5 Switch Operator

The switch operator consists of a series of XOR tests with GOTO statements. The C switch statement requires about 3 assembly statements for each case.

```assembly
027D: MOVF 2A,W
027E: XORLW 01
027F: BTFSC 03.2
0280: GOTO 285
0281: XORLW 03
0282: BTFSC 03.2
0283: GOTO 286
0284: GOTO 287
```

```assembly
0285: INCF 2A,F
0286: DECF 2A,F
```
1.4 Miscellaneous

This section investigates other miscellaneous operators.

1.4.1 Pointers

Pointers in C become simple memory addresses stored in other memory addresses. Again, the assembler uses hexadecimal.

```
................. /* Pointer */
................. pa=&a;
0287: MOVLW 2A
0288: MOVWF 2B
................. *pa;
```

1.4.2 Do-Nothing and Dead Code

First is a do-nothing operation\(^1\) in C. This statement legally compiles but practically does nothing; the resulting assembly code should reflect this. The instruction\(a;\) results in no assembly code.

```
................. /* Do Nothing Code */
................. a; // do nothing
```

Dead code often is compiled out.

```
................. if(0){ // Dead Code
................. a=12;
................. b=12;
................. c=13;
................. }
................. while(0); // Dead Code
```

1.4.3 Including Assembly Code

Including inline assembly code is provided with the `#asm` keyword. Interestingly, an assembly GOTO can refer to a C label!

```
................. /* Assembly Code */
................. #asm
................. NOP
0289: NOP
................. NOP
028A: NOP
................. GOTO 0x0267
```

\(^1\)do-nothing is not no-operation
2 

Printf() Timing

This program times the C printf() function. Intuition suggests that serial data transfer occurs at significantly reduced rates when compared to internal processing data rates. The objective is to use theory and experiments to determine the PIC’s serial port transfer rate.

2.1 Theoretical Data Rate

Theory provides an estimate for serial port data rates. The heart of the printf assembly code is moving the character to the serial port buffer at file 0x19.

\[ \Delta t_{\text{char}} = \frac{\text{bits}}{Baud} = \frac{10}{Baud} \]

For 19200 baud, the time required to transfer one byte is: \( \Delta t_{\text{char}} = \frac{10}{19200} = 0.5\text{ms} \)

2.2 Experimental Data Rate

This section experimentally determines the actual serial data transfer rate. The code is given in printf_time.c in the Code Listings section (p.17). For 14 characters including the linefeed and carriage return, the transfer required 8.3 ms. This is 0.59 ms per byte.

\0Ahello, world\0D

\( \Delta t_{14} = 8.32\text{ms} \) For 100 characters, the time required is 52 ms. This is 0.52 ms.

\( \text{printf(}"12345678901234567890123456789012345678901234567890"	ext{);} \)

Next, the printf() overhead is tested. A single empty printf compiles to no-code. A series of 10 \( \text{printf(}"\"; \) statements require 5.2 ms. The assembly code is:
printf(" ");

A space ' ' character is represented by 0x20 on line 063E. The BTFSS and GOTO statements wait until the transmit buffer is empty — register 0x0C bit 4. Then, the character is moved to file 19, the transmit buffer.

The overhead appears minuscule compared to the serial transfer rate.

This experiment might be biased. The serial data is buffered, so the timer probably stops before last bytes are transferred through the serial cable. Using an oscilloscope, the actual data timing can be determined. Figure 1 gives a total time of 52 ms per 10 printf's.

![Oscilloscope Timing](image)

The printf() function is slow compared to the PIC's internal processing rate. For a 19200 baud serial link and a 20MHz clock, a typical one byte printf requires approximately 26000 times longer than one opcode operation!

### 3 C vs ASM

This part compares C and assembly code fragments. The objective is to become familiar with compiler assembly and hand written assembly and their relationship to C code with respect to efficiency. The code is given in rewrite.c in the Code Listings section (p.19).

---

2The character ‘U’ is used because the ASCII representation is 0x55 or 01010101 binary.
3.1 8bit stored to 16bit

This program adds two 8bit integers and stores the result in a 16bit integer. It was assumed that the problem statement meant: in terms of 16 bit arithmetic, add two 8bit integers. The other possibility is boring and will be ignored — the upper 8 bits will always be zero!

The 16 bit integer is composed of two 8 bit memory addresses at 0x23 and 0x24. The 8 bit integers reside at 0x22 and 0x21. The C and assembly codes are given below. For an unknown reason, a general purpose memory location 0x7A is cleared and added to the 16 bit integer’s upper byte.

\[
\text{c} = (\text{int16})a + (\text{int16})b;
\]

```
007C:  CLRF 28
007D:  CLRF 7A
007E:  MOVF 22,W
007F:  ADDWF 21,W
0080:  MOVWF 23
0081:  MOVF 28,W
0082:  MOVWF 24
0083:  BTFC 03,0
0084:  INCF 24,F
0085:  MOVF 7A,W
0086:  ADDWF 24,F
```

The compiled C code requires 11 assembly operations. The output of this program for inputs \(a=255\) and \(b=123\) is \(c=378\).

Now, assembly code is directly written to perform the same operation. The assembly code is given below.

```
#asm
clrf &d+1 ; clear upper byte
movf a,w ; put a in register
addwf b,w ; add b to register and put in w
btfc 3,0 ; check the carry bit
incf &d+1 ; increment bit 9 of d if carry bit was set
movwf d ; move lower bits to d
#endasm
```

The hand written assembly is 6 operations. Again, the program correctly returns 378.

3.2 For loop

This more complicated program investigated the file-select-register for indirect addressing. The file-select-register allows for variable addressing by creating a pointer FSR to a memory location. The effective dereference operator is the INDF register. The objective of this program was to write a code fragment in C and duplicate the operation with hand-written assembly code. This program freed the author from the bulkiness and waste of direct addressing!
3.2.1 C code and CCS compiler output

The C code is straightforward: a ‘for loop’ sets an array to a constant value. This is effectively a one-line C operation. Values are stored to a changing memory address. C code and the CCS compiler generated assembly are given below. A total of about 116 operations occur for a MAX of 10.

.................... for(index=0; index<MAX; index++) array[index]=1;

*0117: CLRF 21
0118: MOVF 21,W
0119: SUBLW 09
011A: BTFSS 03.0
011B: GOTO 123
011C: MOVLW 22
011D: ADDWF 21,W
011E: MOVWF 04
011F: MOVLW 01
0120: MOVWF 00
0121: INCF 21,F
0122: GOTO 118

3.2.2 Hand Generated ASM

The hand generated ‘for loop’ is given below. Indirect addressing information was found in the 16F876 Datasheet (p.27). Operation is simple. First, the number of iterations is determined from the defined MAX and set into index. Next, the FSR requires a memory address, so a pointer to array was given with movlw (literal move). The loop structure sets the given value for the memory given in the FSR. Next, the FSR is updated. Finally, the loop stops if the decremented index equals zero.

```asm
// Setup Maximum iterations
movlw MAX ; set the maximum array
movwf index ; and set to index

// Pointer to array
movlw array
movwf FSR ; file select register

// Value to set
movlw 2 ; value 2

// Looping
loop:
    movwf INDF ; set value at Indirect register
```

11
4 Assembly Function

This program creates assembly and C functions to bitwise flip a byte. For example, 0x01 would become 0x80. The code is given in asm-function.c in the Code Listings section (p.21). The objective is to become familiar with assembly functions.

The heart of the assembly code is a looped bitwise test `btfsc temp,7,setrrf temp,f`, and rotate `rlf number,f`. The assembly code has 18 lines.

The C code is a looped masked test. A mask and a mirror mask are created for each iteration. The flipped bits are set with:

\[
\text{temp} \text{\|} = ((\text{mask} \& \text{number})==0)? \text{ZEROS} : \text{mirror\_mask};
\]

The program output is given below. For the assembly code, there are 4 initial, 8 loops of 7 and 2 final instructions for a total of 62 instructions. At 0.2 us per instruction, the loop should require about 12.4 us. The ASM actual loop requires 17.6 us. The compiled C code had 52 lines of assembly giving an estimated execution time of 0.8 ms. The actual time was 1.0 ms.

**Assembly Functions:**

**Assembly Code:**
```
1 0 1 0 0 0 1 0
0 1 0 0 0 1 0 1
```
Time = 0.1744000 milliseconds

**C code:**
```
1 0 1 0 0 0 1 0
0 1 0 0 0 1 0 1
```
Time = 1.0464000 milliseconds

For this program, the assembly was about 6 times faster than compiled C.

One small change to the C program improve the efficiency by almost 75%. Remembering that the masks are symmetrical allows for reducing by half the main count loop! Now, the C program requires 0.75 ms, which is still more than 4 times slower than the assembly.

**C code: v2**
```
1 0 1 0 0 0 1 0
0 1 0 0 0 1 0 1
```
Time = 0.7536000 milliseconds
Conclusions

This project investigated assembly programming on the PIC 16F876. The programs experimented with the CCS C compiler and its assembly generation, with the expensive printf() function, inline assembly, and assembly functions. The trivial conclusion is that assembly is faster than C code — up to 10 times faster seems reasonable. However, the largest disadvantage is that assembly quickly lacks code leverage and efficiency due to human constraints. The point appears to be: Use assembly when needed but no more.
/∗ calc_timer.c --- Times various operations ∗
∗
∗ Charles O’Neill ∗
∗ MAE 5483 ∗
∗ Project 3.1 ∗
*/

/* Default PIC Initialization */
#include <16F876.h>
#include <math.h>
#define delay(clock=20000000)
#define fuse HS,NOWDT
#define rs232(baud=19200,parity=N,xmit=PIN_C6,rcv=PIN_C7)

/* Global Variables */
char type[10];

/* Function Prototypes */
void printout(int16 time, int calcs, char* type);
void integer8(void);
void integer16(void);
void floats(void);

/* Main Program for calculation timer */
void main(){

/* Type Declaration Experiment */
int a = 13;
int *pa = &a;
signed int b = 13;
signed int c = -20;
signed int d = 129;
signed int d2 = -128;
signed int d3 = 255;
signed int d4 = 256;
char e = 'e';
char f[4] = {'a', 'b', 'c', 'c'};

/* Inform the user what is happening. */
printf("\n\n\nAssembly Experiment: ");

/* 8 bit Integer */
integer8();

/* 16 bit Integer */
integer16();

/* Floating Point */
floats();

/* Logical Operations */
\n a==1; // compare only (1)
a==13; // compare only (13)
if (a==14) {
    a=2;
} else if (a==4){
a=a;
} else {
    a=3;
}
a=(b==13)? 2 : 3; // conditional operator (13)

/* Infamous GOTO statement */
goto imhere; // Nasty goto
a++; // Should NOT increment a
imhere: --a; // labeled for goto reference

/* Switch */
switch(a) {
    case 1: a++;;
    case 2: a--;;
}

/* Pointer */
pa=&a;
*pa;

/* Do Nothing Code */
a;
if (0) { // Dead Code
    a=12;
    b=12;
    c=13;
}
while(0); // Dead Code
/* Assembly Code */
#asm
NOP
NOP
GOTO 0x0267
GOTO imhere
#endasm

void integer8(void){
    int a = 1;
    int b = 1;
    /* Print Calculation’s Data Type */
    printf("\n\rInteger 8 bit: ");
    /* Addition */
    a = a + 1;
    /* Increment++ */
    a++;
    /* Subtract */
    a = a - 1;
    /* Multiply */
    a = a * b;
    /* Divide */
    a = a / b;
}

void integer16(void){
    int16 a = 1;
    int16 b = 1;
    /* Print Calculation’s Data Type */
    printf("\n\rInteger 16 bit: ");
    /* Addition */
    a = a + 1;
    /* Increment++ */
    a++;
    /* Subtract */
    a = a - 1;
    /* Multiply */
    a = a * b;
    /* Divide */
    a = a / b;
void floats(void)
{
    float a = 1;
    float b = 1;
    /* Print Calculation’s Data Type */
    printf("n\rFloating Point:");

    /* Addition */
    a = a + b;
    /* Subtract */
    a = a - b;
    /* Multiply */
    a = a * b;
    /* Divide */
    a = a / b;
    /* exp() */
    a = exp(b);
    /* log() */
    a = log(b);
    /* sqrt() */
    a = sqrt(b);
    /* cos() */
    a = cos(b);
}

printf_time.c

/* printf_time.c --- Time the printf function */
/* Charles O’Neill */
/* MAE 5483 */
/* Project 4.2 */
*/

/* Default PIC Initialization */
#include <16F876.h>
#define delay(clock=20000000)
#define fuse HS_NOMDT
#define rs232(baud=19200, parity=N, xmit=PIN_C6, rcv=PIN_C7)

/* Global Variables */
#define TIME_SCALE 0.2 // 4/2E6 microseconds per tick

/* Main Program for calculation timer */

void main()
{

/* Type Declarations */
int16 time;
int index=0;

/* Inform the user what is happening. */
printf("n/n/r/n/r/n/r/Printf()Timer: ");

/* Setup Timer */
setup_timer_1(T1INTERNAL | T1_DIV_BY_8);

/* Printf Function Call */
set_timer1(0);
printf("/n/r/n/r/n/r/hello , world\r\n");
time=get_timer1();
printf("/n/r/n/r/n/r/Time=\%9.7f\_milliseconds", ((float)time) * 0.2/1.0E3 * 8.0);

/* Multiple Printf Function Calls */
printf("/n/r/n/r/n/r/");
set_timer1(0);
printf("/12345678901234567890123456789012345678901234567890123456789012345678901234567890\r/n/");
time=get_timer1();
printf("/n/r/n/r/n/r/Time=\%9.7f\_milliseconds", ((float)time) * 0.2/1.0E3 * 8.0);

/* Setup Timer */
setup_timer_1(T1INTERNAL | T1_DIV_BY_1);

/* Printf Function Call */
printf("/n/r/n/r/n/r/");
set_timer1(0);
printf("/n/r/n/r/n/r/");
printf("/n/r/n/r/n/r/");
printf("/n/r/n/r/n/r/");
printf("/n/r/n/r/n/r/");
time=get_timer1();
printf("/n/r/n/r/n/r/Time=\%9.7f\_milliseconds", ((float)time-2) * 0.2/1.0E3);
}
while(1){
    delay_ms(10);
    printf("U");
    printf("U");
    printf("U");
    printf("U");
    printf("U");
    printf("U");
    printf("U");
    printf("U");
    printf("U");
    printf("U");
}

// rewrite.c

/*
 * rewrite .c --- Assembly with the CCS Compiler
 *
 * Charles O' Neill
 * MAE 5483
 * Project 4.3
 */

#include <16F876.h>
#define delay (clock=2000000)
#define fuse HS,NOWDT
#define rs232 (baud=19200, parity=N, xmit=PIN_C6, rcv=PIN_C7)

/* Default PIC Initialization */

/* Global defines and functions */

#define BYTE_LENGTHBITS 8
void forloop( void );
void adding( void );

/* Main Program for calculation timer */

void main(){
    adding();
    forloop();
}
/∗ Adding ∗/  
#include <stdio.h>  

#define MAX 10  
#define FSR 0x04  
#define INDF 0x00  

int index;  
int array[MAX];  

/* C squares 'for loop' */  
for(index=0; index<MAX; index++) array[index]=1;  

/* Generic writeout statement */  
printf("\n\r");  
for(index=0; index<MAX; index++){  
    printf("%d", array[index]);  
}  

void adding(void){  
    /* Variables */  
    int a=255;  
    int b=123;  
    int16 c;  
    int16 d;  

    /* Adding two 8 bit integers into a 16 bit integer */  
    c= (int16)a + (int16)b;  
    printf("a+b=%lu", c);  

    #asm  
    clr f &d+1 ; clear upper byte  
    mov f a,w ; put a in register  
    add w b,w ; add b to register and put in w  
    btfsc 3,0 ; check the carry bit  
    inc f &d+1 ; increment bit 9 of d if carry bit was set  
    movwf d ; move lower bits to d  
    #endasm  
    printf("a+b=%lu", d);  
}  

/* For loop */  
/* For loop */
/* ASM squares 'for loop' */

#asm

// Setup Maximum iterations
movlw MAX ; set the maximum array
movwf index ; and set to index

// Pointer to array
movlw array
movwf FSR ; file select register

// Value to set
movlw 2 ; value 2

// Looping
loop:
    movwf INDF ; set value at Indirect register
    incf FSR,f ; increment pointer address
    decfsz index,f ; Stop loop if decremented index is zero
    goto loop
#endasm

/* Generic writeout statement */
printf("\n\r");
for(index=0; index<MAX; index++){
    printf("%d", array[index]);
}
}

asm-function.c

/* asm.c —— Assembly with the CCS Compiler

*  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

*   Default PIC Initialization

*  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

#include <16F876.h>
#use delay(clock=2000000)
#fuses HS,NOWDT
```c
#define TIME_SCALE 5   // 2E6/4 ticks per microsecond

void printout(int16 time, int calcs, char* type);

int testasm(int number){
    int count=0;
    #asm
    loop:
        incf count,f;
        decfsz number,f;
        goto loop;
        movf count,w;  // move count into w register
        movwf _return_;  // return w
    #endasm
}

int testasmC(int number){
    int count=0;
    while(--number){
        count++;
    }
}

void main(){
    int a;
    /* Inform the user what is happening. */
```
printf("\n\n\n\n\n\n\n\nAssembly Functions:");

/* Setup Timer */
setup_timer_1(T1_INTERNAL | T1_DIV_BY_I);

a = testasm(52);
printf("\n%d", a);
a = testasmC(52);
printf("\n%d", a);