Lesson 9
Assembler Directives

Overview

Introduction	In this lesson, we will examine a number of assembler directives that we will not cover elsewhere in the course.
In this section	Following is a list of topics in this section:

Number Formats

Introduction

Up until now, we have always represented numbers as a letter, either H, D, O or B, followed by a string of digits enclosed in apostrophes. This is the number format recommended by Microchip, but it is not the only format supported by the assembler. In fact, as you look at other programs, you will discover that it isn't even a very common format. Here we will look at a number of other formats.

Binary Numbers

During this course, binary numbers have always been represented in the Microchip recommended format:

```
B'01111010'
```

According to the assembler documentation, this is the only way to represent binary constants. (However, the assembler documentation is incomplete in some other areas!)

Octal Numbers

Octal numbers have been represented with the letter O, as in

```
O'172'
```

However, whenever a string begins with a number, the assembler assumes that it is a numeric constant. Most programmers add a leading zero to clarify that this is a number, but to the assembler it is only necessary that the leading character be from 0 to 9. Notice that in many other languages, a leading zero implies an octal number. This is not the case in the PIC assembler. The trailing character of the constant may define the radix. In the case of octal, a trailing 'O' tells the assembler that the number is octal:

```
0172o
```

The 'O' may be upper or lower case, however, lower case is far easier to differentiate from the character zero for most fonts.

ASCII characters

Although not specifically a 'number', it is often helpful to represent the value of an ASCII character. This can be done, following the Microchip convention, by using the letter A:

```
A'z'
```

In many cases, the 'A' isn't actually required; the ASCII character can simply be surrounded with apostrophes:

```
'z'
```

Continued on next page

Number Formats, Continued

Decimal Numbers

Decimal numbers have been represented with the letter D:

```
D'122'
```

However, the assembler can also recognize that a number is a decimal number if it is prefixed with a decimal point:

```
.122
```

This is probably the more common representation in programs you will find on the net.

Hexadecimal Numbers

So far, the letter H, in the recommended Microchip style, has been used for hexadecimal numbers:

```
H'7a'
```

However, a string beginning with a digit and ending with the letter H is also interpreted as a hex number. In fact, probably the most common form of hex number you will see in programs is something like:

```
07ah
```

But the assembler will also accept the C style hex number:

```
0x7a
```

Finally, to round out the selection, the assembler has a default radix that it applies to any string beginning with a digit, and by default, the default radix is hex. So the following is the same as the above examples:

```
7a
```

The Radix directive

While the default radix is hexadecimal, the default can be changed with the radix directive. Thus:

```
        radix   dec
A       equ     122
```

Is equivalent to

```
A       equ     D'122'
```

You may switch between radices as often as you want, but be advised that it can be confusing for someone reading your program. The choices for radix are dec, oct, and hex. This author recommends that you never switch radices, and always explicitly state the radix for every constant. However, when you look at programs written by others, you may find every possible format.

Conditional Assembly

Introduction

Sometimes it can be useful to have several programs that are almost identical. For example, there may be a debug version of a program as well as a normal version. Rather than keeping two, almost identical, programs, the assembler can be directed to include parts of the code sometimes, and other parts at other times.

The #define directive

The #define directive allows us to associate a text string with a name. Whenever the assembler sees the name, it will replace it with the text string:

```
#define  length   .20
```

The text string is actually optional, it is perfectly permissible to define a name that is associated with an empty string. In fact, this is probably more common:

```
#define  debug
```

ifdef-else-endif

The `ifdef` directive begins a block of code which will be executed if a symbol is defined. The `else` directive begins an alternate block of code. Finally, the `endif` directive ends the conditional part of the code.

One way to use this is to have special code for debugging. For example, when debugging timing loops, it can be very tedious to step through them with the simulator. The number of iterations can be conditionally changed for testing:

```
#define debug
            ifdef       debug
count       equ         H'03'
            else
count       equ         H'7a'
            endif
```

Notice that the `#define` directive starts in column one; the others start in the opcode column like most other directives.

It is important to recognize that this if-then-else logic is evaluated at assembly time. This logic will not be interpreted within the PIC; only one path through the logic will actually end up in the code loaded into the PIC.

There is also an almost identical if-then-else, however, for our purposes, this is quite uncommon compared to the ifdef logic.

Controlling Assembler Messages

Introduction

Previously in this course, the `errorlevel` directive has been used to turn off a specific message. It can be used it to suppress any specific message by prefixing the message number with a minus sign:

```
errorlevel    -302
```

and can turn the message back on by using a plus sign:

```
errorlevel    +302
```

Message Levels

There are actually three categories of messages generated by the assembler; messages, warnings and errors. "Messages" are the least severe, "errors" the most.

Messages less severe than a specified level may be suppressed with the `errorlevel` directive as well.

```
errorlevel    2
```

will suppress messages and warnings, while

```
errorlevel    1
```

will suppress only messages.

```
errorlevel    0
```

is the default, and will cause all the assembler messages to be displayed.

Creating your own messages

While we normally do not want more errors to be displayed, the assembler does provide a way for us to generate our own messages. The `messg` directive allows us to insert a string into the assembler's output:

```
messg    "Watch out – debug code"
```

Similarly, an actual error, rather than just a message, can be created which stops the assembly:

```
error    "Parameters out of range"
```

This is primarily useful in macros (which will not be covered right now) and in conditional assembly.

```
      #define debug
                  ifdef       debug
                  messg       "**Watch out, debug code**"
      count       equ         H'03'
                  else
      count       equ         H'7a'
                  endif
```

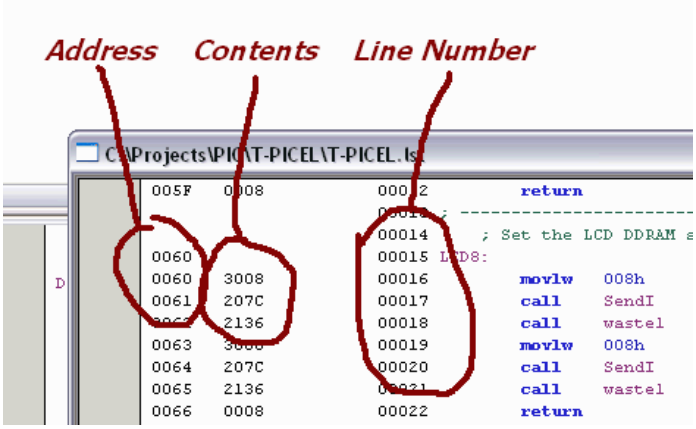
The Listing File

Introduction

Whenever you do an assembly, the assembler creates a listing file. You can look at this listing file through MPLAB, or you can open it with WordPad or your favorite test editor. The listing file has some interesting information in it. Many of the assembler directives of interest are aimed at controlling the listing file.

What's in the listing

The listing file is useful for is seeing just what code the assembler is generating. Each line that generates code has the line number within your source, as well as the value of the code generated:



Toward the bottom of the file is a listing of all the symbols you have used and their values:

```
SYMBOL TABLE
  LABEL                             VALUE

AD9850_0                          00000047
AD9850_1                          00000048
AD9850_2                          00000049
AD9850_3                          0000004A
```

Still farther down is a map of your program memory use:

```
MEMORY USAGE MAP ('X' = Used,  '-' = Unused)

0000 : X----XXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
0040 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
0080 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
00C0 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
0100 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
0140 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
0180 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
01C0 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
0200 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
0240 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
0280 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
02C0 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX
0300 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXX---- ----------------
2000 : -------X-------- ---------------- ---------------- ----------------

All other memory blocks unused.

Program Memory Words Used:   808
Program Memory Words Free:   216
```

Continued on next page

The Listing File, Continued

What's in the listing (continued)

Finally, there is a count of all the messages, including those you may have suppressed with the `errorlevel` directive:

```
Errors   :     0
Warnings :     0 reported,     0 suppressed
Messages :     0 reported,     6 suppressed
```

Page Titles

At the top of each page the listing shows the assembler (not MPLAB) version and assembly date, as well as the page number. The assembler can also be directed to include a title and subtitle to be shown at the top of each page.

```
 MPASM 03.50 Released                                T-PICEL.ASM   2-3-2004
PIC-EL Test Suite                        John J. McDonough - 23-Dec-03
LCD Support Routines
```

This is controlled with the `title` and `subtitle` directives:

```
title       "PIC-EL Test Suite"
subtitle    "LCD Support Routines"
```

Additional listing control

The `list` directive can turn the listing on and off (useful for suppressing printing of uninteresting include files), as well as control various aspects of the listing. Perhaps the most useful is to control the tab size. MPLAB defaults to 4 character tabs, but the listing defaults to 8 character tabs. This is changed with the "`b=`" phrase in the `list` directive. Other useful features are the page length (`n=`) and the width (`c=`):

```
list      b=4,c=132,n=60
```

Finally, there is a `page` directive that forces a new page. This can be useful to start a routine, or group of routines, at the top of the page:

```
page
```

Wrap Up

Introduction	In this lesson, we have looked at a number of assembler directives that allow us to control what the assembler processes and how to control the appearance of the listing file.
Coming Up	In the next lesson, we will install FPP, the program we will use to program the PIC with our software, and we will use FPP to test the programmer portion of our PIC-EL.