MPLAB® XC8 PIC Assembler User's Guide for Embedded Engineers

Notice to Development Tools Customers

Important:
All documentation becomes dated, and Development Tools manuals are no exception. Our tools and documentation are constantly evolving to meet customer needs, so some actual dialogs and/or tool descriptions may differ from those in this document. Please refer to our website (www.microchip.com/) to obtain the latest version of the PDF document.

Documents are identified with a DS number located on the bottom of each page. The DS format is DS<DocumentNumber><Version>, where <DocumentNumber> is an 8-digit number and <Version> is an uppercase letter.

For the most up-to-date information, find help for your tool at onlinedocs.microchip.com/.

Table of Contents

1. Preface

1.1 Conventions Used in This Guide

The following conventions may appear in this documentation:

Table 1-1. Documentation Conventions

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	*MPLAB® IDE User's Guide*
	Emphasized text	...is the *only* compiler...
Initial caps	A window	the Output window
	A dialog	the Settings dialog
	A menu selection	select Enable Programmer
Quotes	A field name in a window or dialog	"Save project before build"
Underlined, italic text with right angle bracket	A menu path	*File>Save*
Bold characters	A dialog button	Click **OK**
	A tab	Click the **Power** tab
N'Rnnnn	A number in verilog format, where N is the total number of digits, R is the radix and n is a digit.	4'b0010, 2'hF1
Text in angle brackets < >	A key on the keyboard	Press <Enter>, <F1>
Courier New font:		
Plain Courier New	Sample source code	`#define START`
	Filenames	`autoexec.bat`
	File paths	`c:\mcc18\h`
	Keywords	`_asm, _endasm, static`
	Command-line options	`-Opa+, -Opa-`
	Bit values	`0, 1`
	Constants	`0xFF, 'A'`
Italic Courier New	A variable argument	*`file.o`*, where *`file`* can be any valid filename
Square brackets [ ]	Optional arguments	`mcc18 [options] file [options]`
Curly brackets and pipe character: { | }	Choice of mutually exclusive arguments; an OR selection	`errorlevel {0|1}`
Ellipses...	Replaces repeated text	`var_name [, var_name...]`
	Represents code supplied by user	`void main (void) { ... }`

1.2 Recommended Reading

This user’s guide describes the use and features of the MPLAB XC8 PIC Assembler. The following Microchip documents are available and recommended as supplemental reference resources.

MPLAB® XC8 PIC Assembler User's Guide
This user’s guide describes the use and features of the MPLAB XC8 PIC Assembler.

MPLAB® XC8 PIC Assembler Migration Guide
This guide is for customers who have MPASM projects and who wish to migrate them to the MPLAB XC8 PIC assembler. It describes the nearest equivalent assembler syntax and directives for MPASM code.

MPLAB® XC8 C Compiler Release Notes for PIC MCU
For the latest information on changes and bug fixes to this assembler, read the Readme file in the docs subdirectory of the MPLAB XC8 installation directory.

Development Tools Release Notes
For the latest information on using other development tools, read the tool-specific Readme files in the docs subdirectory of the MPLAB X IDE installation directory.

2. Introduction

This guide shows and describes example assembly programs that can be built with the MPLAB® XC8 PIC Assembler (PIC Assembler) for Baseline, Mid-range, and PIC18 device families.

The examples shown in this document bring together various programming concepts, assembler directives and operators, as well as command-line options, which you can read about in more detail in the *MPLAB® XC8 PIC Assembler User's Guide*.

If you are migrating programs from the Microchip MPASM™ Assembler to the PIC Assembler, this guide in conjunction with the *MPLAB® XC8 PIC Assembler Migration Guide* will be of great assistance.

3. A Basic Example For PIC18 Devices

As an introduction to the PIC Assembler, consider the following example of a complete assembly program. It is written for a PIC18F47K42 device, but much of this example is also applicable for other devices. The code performs the mundane task of repeatedly reading the value on PORTA and recording the highest value that has been read. Aspects of this program are discussed in the sections that follow.

Note: This code example performs manual masking of instruction operand addresses to avoid fixup overflow errors. You can alternatively have the linker automatically truncate operand values when building, as discussed in 4.3. Working with Data Banks, so that the `BANKMASK()` and `PAGEMASK()` macros used in this example are not required.

A Basic PIC18 Example

```
/* Find the highest PORTA value read, storing this into
   the object max */

PROCESSOR 18F47K42

#include <xc.inc>

CONFIG "FEXTOSC = OFF"        // Ext Oscillator Selection->Oscillator not enabled
CONFIG "RSTOSC = HFINTOSC_1MHZ"   // Reset Osc Select->HFINTOSC,HFFRQ=4MHz,CDIV=4:1
CONFIG "CLKOUTEN = OFF"       // Clock out Enable bit->CLKOUT function is disabled
CONFIG "PR1WAY = ON"          // PRLOCKED 1-Way Set->PRLOCK cleared/set only once
CONFIG "CSWEN = ON"           // Clock Switch Enable->Writing to NOSC & NDIV allowed
CONFIG "FCMEN = ON"           // Fail-Safe Clock Monitor->Clock Monitor enabled

CONFIG "MCLRE = EXTMCLR"      // If LVP=0, MCLR pin is MCLR; LVP=1, RE3 pin is MCLR
CONFIG "PWRTS = PWRT_OFF"     // PWRT is disabled
CONFIG "MVECEN = OFF"         // Vector table isn't used to prioritize interrupts
CONFIG "IVT1WAY = ON"         // IVTLOCK bit can be cleared and set only once
CONFIG "LPBOREN = OFF"        // ULPBOR disabled
CONFIG "BOREN = SBORDIS"      // Brown-out Reset enabled, SBOREN bit is ignored
CONFIG "BORV = VBOR_2P45"     // Brown-out Reset Voltage (VBOR) set to 2.45V
CONFIG "ZCD = OFF"            // ZCD disabled, enable by setting ZCDSEN in ZCDCON
CONFIG "PPS1WAY = ON"         // PPS locked after clear/set cycle
CONFIG "STVREN = ON"          // Stack full/underflow will cause Reset
CONFIG "DEBUG = OFF"          // Background debugger disabled
CONFIG "XINST = OFF"          // Extended Instruction Set disabled

CONFIG "WDTCPS = WDTCPS_31"   // Divider ratio 1:65536; software control of WDTPS
CONFIG "WDTE = OFF"           // WDT Disabled; SWDTEN is ignored
CONFIG "WDTCWS = WDTCWS_7"    // window open 100%; software control
CONFIG "WDTCCS = SC"          // Software Control

CONFIG "BBSIZE = BBSIZE_512"  // Boot Block size is 512 words
CONFIG "BBEN = OFF"           // Boot block disabled
CONFIG "SAFEN = OFF"          // SAF disabled
CONFIG "WRTAPP = OFF"         // Application Block not write protected
CONFIG "WRTB = OFF"           // Configuration registers not write-protected
CONFIG "WRTC = OFF"           // Boot Block (000000-0007FFh) not write-protected
CONFIG "WRTD = OFF"           // Data EEPROM not write-protected
CONFIG "WRTSAF = OFF"         // SAF not Write Protected
CONFIG "LVP = ON"             // LV programming enabled, MCLR pin, MCLRE ignored

CONFIG "CP = OFF"             // PFM and Data EEPROM code protection disabled

GLOBAL max              ;make this global so it is watchable when debugging

;objects in common (Access bank) memory
PSECT udata_acs
max:
    DS         1        ;reserve 1 byte for max
tmp:
```

```
    DS         1        ;1 byte for tmp

;this must be linked to the reset vector
PSECT resetVec,class=CODE,reloc=2
resetVec:
    goto       main

/* find the highest PORTA value read, storing this into
   the object max */
PSECT code
main:
    ;set up the oscillator
    movlw      0x62
    movlb      57
    movwf      BANKMASK(OSCCON1),b
    movlw      2
    movwf      BANKMASK(OSCFRQ),b
    movlb      58
    clrf       BANKMASK(ANSELA),b  ;select digital input for port A

    clrf       max,c               ;starting point
loop:
    movff      PORTA,tmp           ;read and store the port value
    movf       tmp,w,c             ;is this value larger than max?
    subwf      max,w,c
    bc         loop                ;no - read again
    movff      tmp,max             ;yes - record this new high value
    goto       loop                ;read again

    END        resetVec
```

3.1 Comments

There are several styles of comments that can be used in assembly source code.

Assembler comments consist of a semi-colon, `;`, followed by the comment text. These can be placed on a line of their own, such as the comment:

```
;objects in common (Access bank) memory
```

They can also be at the end of a line containing an instruction or directive, as shown in the line:

```
    movff    PORTA,tmp     ;read the port
```

C-style comments can be used in assembly source code if the assembly source file is preprocessed. To run the preprocessor over an assembly source file, name the file using a `.S` extension (upper case), or use the `-xassembler-with-cpp` option when you build.

Single-line C-style comments begin with a double slash sequence, `//`, followed by the comment text, as seen in the line:

```
CONFIG PWRTS=PWRT_OFF    // PWRT is disabled
```

Multi-line C-style comments begin with slash-star, `/*`, and terminate with a star-slash sequence, `*/`. This is the only comment style that allow the comment text to run over multiple lines. One is shown in the example:

```
/* find the highest PORTA value read, storing this into
   the object max */
```

3.2 Configuration Bits

The configuration bits of your device must always be set to appropriate values. It is unlikely your program will execute correctly, or at all, if these are not valid. You should consult your device data sheet to ensure that you understand the function of each configuration setting and the appropriate value that should be used.

You can use the MPLAB X IDE to assist in the creation of the code necessary to set the configuration bits, but any code it produces must be copied into a source file that is a part of your project.

In this chapter's example, the configuration bits have been specified using the PIC Assembler's `CONFIG` directive, for example:

```
CONFIG "WDTE=OFF"    // WDT Disabled; SWDTEN is ignored
```

As shown in the example code, you typically provide setting-value pairs to these directives, setting each configuration bit to the desired state, but an entire configuration word can also be programmed using one directive, if required. The `CONFIG` directive can also be used to set the device's IDLOC bits, where relevant.

Note that the setting-value pairs in the example code are quoted. This is to ensure that there can be no interaction between any predefined preprocessor macros (when the preprocessor is enabled) and the `CONFIG` directive data. It is recommended that quotes always be used.

If you are not using the MPLAB X IDE, the configuration bit settings and values associated with each device can be determined from an HTML guide. Open either the `pic18_chipinfo.html` guide for PIC18 devices or the `pic_chipinfo.html` guide for all other devices. These are located in the `docs` directory in the MPLAB XC8 C Compiler distribution that contains your PIC Assembler. Click the link to your target device and the page will show you the settings and values that are appropriate for your directives. Note that the usage examples shown in these HTML guides demonstrate the `#pragma config` directive used with C code as well as the `CONFIG` directive, for the PIC Assembler (shown above). Ensure you follow the “PIC-AS config Usage” examples.

3.3 Include Files

Just as with C source code, assembly code can include the content of other files.

The best way to include files into your source code is with the `#include` directive. As this is a preprocessor directive, the preprocessor must be run over the source file for it to be executed. To do this, use a (capital) `.S` extension with the source file name or use the `-xassembler-with-cpp` option.

The preprocessor will search for included files specified in angle brackets, `< >`, from standard directories in the assembler distribution. Searches for file names that are quoted, `" "`, will begin in the current working directory, then in the standard directories. You can specify additional search paths by using the `-I` option.

All but the most trivial of assembly programs will need to include the `<xc.inc>` include file, as shown in this chapter's example by the line:

```
#include <xc.inc>
```

This include file is provided by the PIC Assembler and allows your source code to access special function registers, predefined psects and macros, and other device features. Do *not* include device-specific include files (for example, do *not* include files like `pic18lf8720.inc`) or maintain your own version of these files, as their content might change in future versions of the assembler.

3.4 Commonly Used Directives

There are several assembler directives that are typically used in each module. These have been used in the example code for this chapter and are discussed here.

Processor Directive

The `PROCESSOR` directive is not mandatory but should be used if the assembler source in the module is only applicable to one device.

The `-mcpu` option always specifies which device the code is being built for. If the `PROCESSOR` directive has been used and there is a mismatch between the device specified in that directive and in the option, an error will be triggered. The line of code:

```
PROCESSOR 18F47K42
```

shows the PIC18F47K42 device being specified. With this in place, the code cannot be built for any other device.

If your code can be built for multiple devices, the `PROCESSOR` directive should be omitted. You can instead make sections of your code specific to the target device by using the preprocessor's conditional inclusion features and preprocessor macros predefined by the PIC Assembler. For example:

```
#ifdef _18F47K40
  movwf LATE
#endif
```

will assemble the `movwf` instruction only for PIC18F47K40 devices. The `_18F47K40` preprocessor macro is one that is automatically defined by the assembler, based on the device selected by the `-mcpu` option. A full list of predefined macros is available in the *MPLAB® XC8 PIC Assembler User's Guide*.

End Directive

Use of the `END` directive is not mandatory, but signifies an end to the source code in that module. There can be no further lines of source present after an `END` directive, even blank ones.

If you use one or more `END` directives in your program, one (and only one) of those directives should specify the program's entry point label to prevent an assembler warning being generated. This has been done in the last line of the example program:

```
END resetVec
```

3.5 Predefined Psects

All code, data objects, or anything that is to be linked into the device memory must be placed in a psect.

Psects—short for program sections—are containers that group and hold related parts of the program, even though the source code for these parts might not be physically adjacent in the source file, or may even be spread over several modules. They are the smallest entities positioned by the linker into memory.

To place code or objects into a psect, precede their definition with a `PSECT` directive and the name of the psect you wish to use. The first time a psect is used in a module, you must define the psect flags that are relevant for the psect. These flags control which memory space the psect will reside in and further refine how the psect is linked.

To assist with code migration and development, several psects are predefined with appropriate flags by the PIC Assembler and are available once you include the `<xc.inc>` file into your source. A full list of these psects can be found in the *MPLAB® XC8 PIC® Assembler User's Guide*.

Attention: You must include the `<xc.inc>` include file to be able to use the predefined psects supplied by the assembler.

In this chapter's example, the predefined `code` psect was used to hold most of the program's code. You can see it being specified in the lines:

```
PSECT code
main:
    ;set up the oscillator
    movlw      0x62
    ...
```

thereby placing the `main` label and the instructions that follow into the `code` psect. The `udata_acs` psect has been used to place the `max` and `tmp` labels and the reserved memory they are associated with in the PIC18 Access Bank memory, as shown in the lines:

```
PSECT udata_acs
max:
    DS          1           ;reserve 1 byte for max
tmp:
    DS          1           ;1 byte for tmp
```

The configuration data also needs to be inside a psect so that it will appear in the HEX file at the appropriate address; however, the `CONFIG` directives will automatically ensure that their output is in a psect that will be linked to the required location, so you should not precede them with a `PSECT` directive in your code. You can see where the configuration data is linked in the map file, which is illustrated in section 3.7. Building the Example.

One advantage of using predefined psects is that they are already associated with a suitable linker class, so they will be automatically linked into an appropriate region of memory without you having to stipulate any linker options when you build your project.

3.6 User-defined Psects for PIC18 Devices

Rather than use the PIC Assembler's predefined psects, you will need to define your own unique psects to have code or objects linked to special locations. These psects can then be linked to the required address without affecting the placement of the remainder of your code or data.

In this chapter's example, the program's entry point consists of a short segment of code that is to be executed immediately after a Reset. This code, therefore, must be linked at the Reset vector, address 0.

To place code or objects into a psect, use a `PSECT` directive and the name of the psect you wish to use. The first time a psect is used in a module, you must define the psect flags that are relevant for the psect. These flags control which memory space the psect will reside in and further refine how the psect is linked. The flags can be omitted on subsequent uses of the `PSECT` directive for the same psect. The flags on psects with the same name but in other modules must agree.

In the example, the psect that was used to hold the Reset code was defined as follows, but note that the flags used with this psect are only relevant for PIC18 devices.

```
PSECT resetVec,class=CODE,reloc=2
resetVec:
    goto main
```

The psect name being defined and selected by this directive is `resetVec`. The label and `goto` instruction that follow will be part of this psect. Note that psect names are case sensitive.

A psect is typically associated with a linker class through the `class` psect flag. In the above example, the `resetVec` psect has been associated with a linker class called `CODE`, which is a class predefined by the assembler. A class defines one or more ranges of memory in which the psect can be linked, if desired. As the `resetVec` psect in this example must be explicitly linked to an absolute address (the Reset vector), the class association is not actually necessary, but it is common to always specify a psect's class, if only to make the psect's listing in the map file easier to find. Psects associated with a class can still be placed at arbitrary locations using a linker option, in which case, the class's memory ranges are ignored. The predefined classes are listed in the *MPLAB® XC8 PIC® Assembler User's Guide*.

The `reloc` flag in the `resetVec` psect ensures that the start of the psect is aligned to an address that is a multiple of the flag's value. It is critical that this flag be set to 2 for any PIC18 psects that hold executable code, as instructions must be word aligned on these devices. Set it to 1 (the default value if no `reloc` flag is specified) for psects holding instructions for other devices. With this flag in place, the linker will issue an error if the linker option you use to place the psect, which is shown in section 3.7. Building the Example, requests an incompatible address. For other situations where a psect has allocated memory anywhere in a linker class range, the linker will choose an address that is aligned to the specified value.

Any of the predefined psects used to hold instructions, such as `code`, specify the relevant `reloc` value for the device you are using.

3.7 Building the Example

One or more assembly source files can be built with a single execution of the `pic-as` assembler driver.

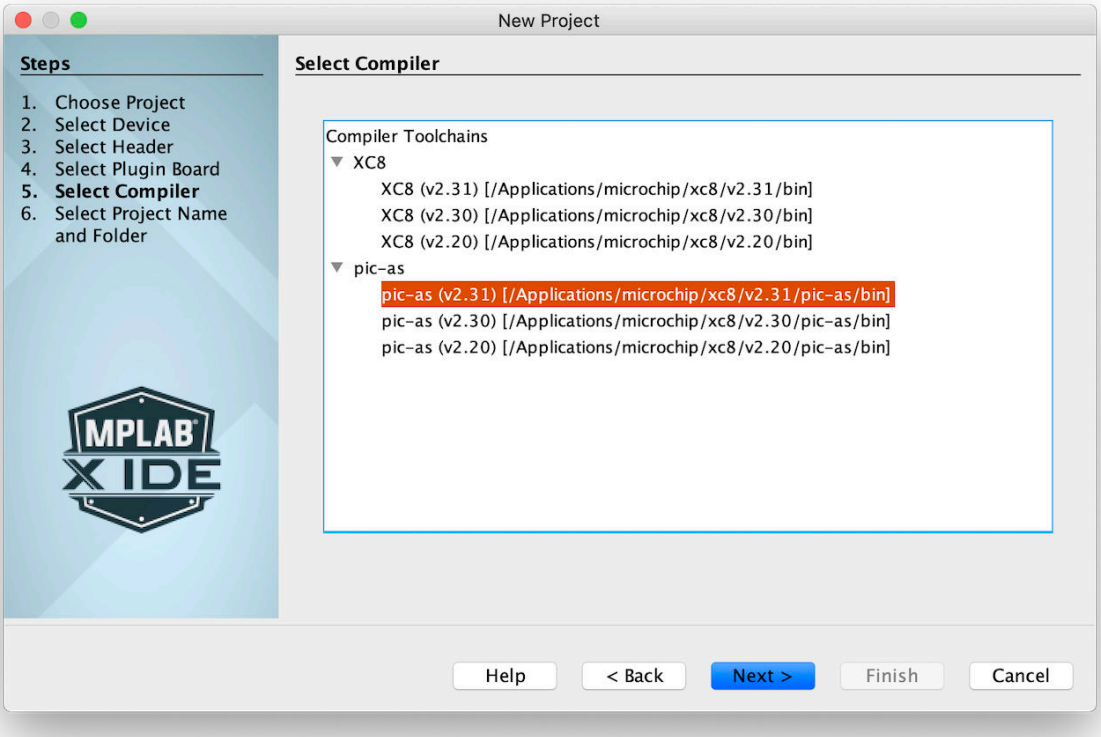
If the entire source code shown in section 3. A Basic Example For PIC18 Devices was saved in a plain text file, called `readPort.S`, for example, it could be built using the command below. Note that the file name uses an upper case `.S` extension so that the file is preprocessed before any other processing.

```
pic-as -mcpu=18f47k42 readPort.S
```

This command will produce (among other files) `readPort.hex` and `readPort.elf`, which can be used by the IDE and debugger tools to execute and debug the code. Such a command assumes that `pic-as` is in your console's search path. If that is not the case, specify the full path to `pic-as` when you run the application.

If you prefer to build in the MPLAB X IDE, create a project in the usual way. When asked to select the compiler, choose the required version of the `pic-as` toolchain. The PIC Assembler is shipped with the MPLAB XC8 C Compiler, but make sure you choose the assembler tool, as shown in the following image.

Figure 3-1. Selecting the PIC Assembler for MPLAB X IDE projects



Although the above command line (or default IDE project settings) will build the example code with no error, there are, however, some additional options that need to be specified before the code will run. To see why, build the code again using the `-Wl` driver option to request a map file, as shown below. (How to specify additional options in an IDE project are given at the end of this section.)

```
pic-as -mcpu=18f47k42 -Wl,-Map=readPort.map readPort.S
```

Open `readPort.map` and look for the special `resetVec` psect that was defined in the code. You will see something similar to the following:

```
TOTAL           Name                 Link     Load   Length     Space
        CLASS   CODE
                resetVec            1FFDE    1FFDE        4         0
                code                1FFE2    1FFE2       1E         0
```

The `resetVec` psect will be present in several locations, but you can easily find it under the `CODE` class, with which it was associated. Notice that it was linked to 1FFDE, not address 0, as we require, because the linker received no explicit placement instructions and simply linked it to a free location in the memory defined by the `CODE` class. Note that it did, however, honor the psect's `reloc=2` flag and linked it to an address that was a multiple of 2.

You can see the definition (`-A` linker option) for the `CODE` (and other) linker classes at the top of the map file as part of the linker options. For this device, the definition for the `CODE` class is:

```
-ACODE=00h-01FFFFh
```

which shows that this class represents one large chunk of memory from address 0 through 0x1FFFF.

Build the source file again, this time use the `-Wl` driver option to pass a `-p` option to the linker. In the command line below, the option explicitly places the `resetVec` psect at address 0.

```
pic-as -mcpu=18f47k42 -Wl,-Map=readPort.map -Wl,-presetVec=0h readPort.S
```

The content of the map file will now look similar to the following, showing that the reset code is in the correct location.

```
TOTAL           Name                 Link     Load   Length     Space
        CLASS   CODE
                resetVec               0        0        4         0
                code                1FFE2    1FFE2       1E         0
```

To confirm the linker's handling of the `reloc` psect flag, you can try 'accidentally' linking the `resetVec` psect to an address that is not a multiple of 2 (for example address 0x1) using the above option.

Notice also in the map file that the data specified by the `CONFIG` directives were placed into a psect called `config`, which was linked to the correct address in the hex file for this device.

```
TOTAL           Name                 Link     Load   Length     Space
...
        CLASS   CONFIG
                config                       300000   300000         A         4
```

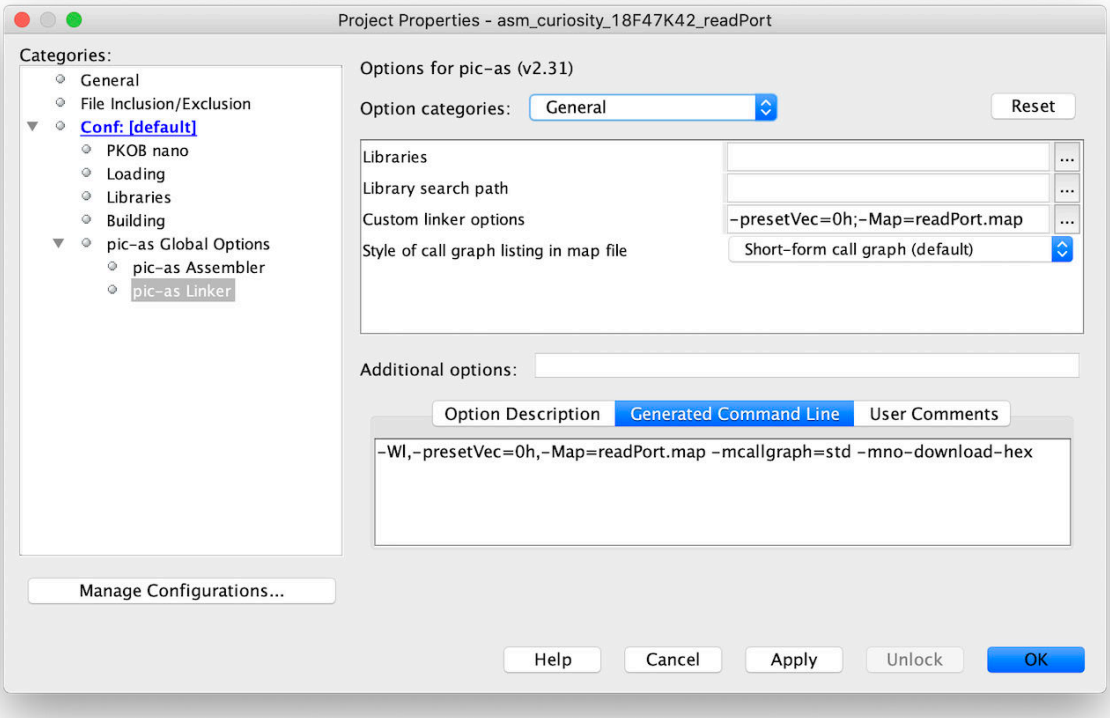
You can request an assembly list file be created with the option `-Wa,-a`, so you can explore the generated code, for example:

```
pic-as -mcpu=18f47k42 -Wl,-Map=readPort.map -Wl,-presetVec=0h -Wa,-a readPort.S
```

This option creates the list file with the same base name as the first source file present in the command line and with the extension, `.lst`. The above command will generate a list file called `readPort.lst`.

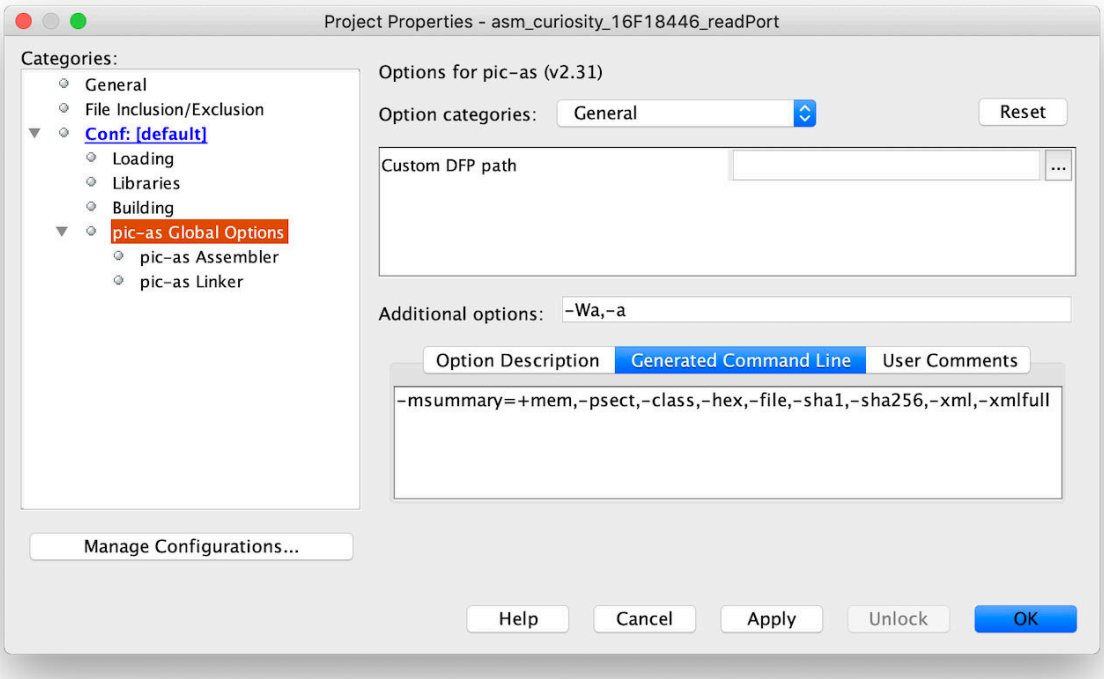
If you are building in the IDE, add additional linker options (those specified above using the driver option `-Wl,`*`linkerOption`* above) to the **Custom linker options** field in the **pic-as Linker** > **general** category in the Project Properties dialog, as shown in the following image. When adding option to this field, do not include the leading `-Wl,`, as this prefix is added in by the IDE.

Figure 3-2. Specifying additional options to be processed by the linker in PIC Assembler projects



In the IDE, specify additional options to the assembler (such as the `-Wa,-a`) in the **Additional options** field in the **pic-as Global Options** category, as shown in the following image.

Figure 3-3. Specifying additional options to be processed by the assembler in PIC Assembler projects



4. A Basic Example For Mid-range Devices

The following example of a complete assembly program is written for a PIC16F18446 device, although much of this example is also applicable for other devices. It performs the mundane task of repeatedly reading the value on PORTC and recording the highest value that has been read. This example follows on from information presented in section 3. A Basic Example For PIC18 Devices and aspects of the code are discussed in the sections that follow.

A Basic PIC16 Example

```
/*
 * Find the highest PORTC value read, storing this into the object max
 */

PROCESSOR 16F18446

#include <xc.inc>

CONFIG "FEXTOSC = OFF"      // External Oscillator mode->Oscillator not enabled
CONFIG "RSTOSC = HFINT1"    // Power-up default value for COSC->HFINTOSC (1MHz)
CONFIG "CLKOUTEN = OFF"     // CLKOUT disabled; i/o or osc function on OSC2
CONFIG "CSWEN = ON"         // Clock Switch Enable->Writing to NOSC and NDIV allowed
CONFIG "FCMEN = ON"         // Fail-Safe Clock Monitor Enable->FSCM timer enabled

CONFIG "MCLRE = ON"         // Master Clear->MCLR pin is Master Clear function
CONFIG "PWRTS = OFF"        // Power-up Timer Enable bit->PWRT disabled
CONFIG "LPBOREN = OFF"      // Low-Power BOR enable bit->ULPBOR disabled
CONFIG "BOREN = ON"         // Brown-out reset enable->Enabled, SBOREN bit ignored
CONFIG "BORV = LO"          // Brown-out Reset Voltage Selection->VBOR set to 2.45V
CONFIG "ZCDDIS = OFF"       // Zero-cross detect disable->disabled at POR
CONFIG "PPS1WAY = ON"       // PPSLOCK cleared/set only once
CONFIG "STVREN = ON"        // Stack Over/underflow causes reset

CONFIG "WDTCPS = WDTCPS_31" // WDT Period Select->Ratio 1:65536; software control
CONFIG "WDTE = OFF"         // WDT operating mode->WDT Disabled, SWDTEN is ignored
CONFIG "WDTCWS = WDTCWS_7"  // WDT Window Select->always open; software control
CONFIG "WDTCCS = SC"        // WDT input clock selector->Software Control

CONFIG "BBSIZE = BB512"     // Boot Block Size Selection->512 words boot block size
CONFIG "BBEN = OFF"         // Boot Block Enable bit->Boot Block disabled
CONFIG "SAFEN = OFF"        // SAF Enable bit->SAF disabled
CONFIG "WRTAPP = OFF"       // Application Block not write protected
CONFIG "WRTB = OFF"         // Boot Block not write protected
CONFIG "WRTC = OFF"         // Configuration Register not write protected
CONFIG "WRTD = OFF"         // Data EEPROM NOT write protected
CONFIG "WRTSAF = OFF"       // Storage Area Flash not write protected
CONFIG "LVP = ON"           // LV Programming Enabled, MCLR/Vpp pin is MCLR

CONFIG "CP = OFF"           // UserNVM Program memory code protection->Disabled

skipnc  MACRO
    btfsc      CARRY
ENDM

;objects in bank 0 memory
PSECT udata_bank0
max:
    DS         1                ;reserve 1 byte for max
tmp:
    DS         1                ;reserve 1 byte for tmp

PSECT resetVec,class=CODE,delta=2
resetVec:
    PAGESEL    main             ;jump to the main routine
    goto       main
```

```
/* find the highest PORTA value read, storing this into
   the object max */
PSECT code
main:
    ;set up the oscillator
    movlw      0x62
    movlb      17
    movwf      OSCCON1
    movlw      2
    movwf      OSCFRQ
    PAGESEL    loop                ;ensure subsequent jumps are correct
    BANKSEL    max                 ;starting point
    clrf       BANKMASK(max)
    BANKSEL    ANSELC
    clrf       BANKMASK(ANSELC)    ;select digital input for port C
loop:
    BANKSEL    PORTC               ;read and store port value
    movf       BANKMASK(PORTC),w
    BANKSEL    tmp
    movwf      BANKMASK(tmp)
    subwf      max^(tmp&0ff80h),w  ;is this value larger than max?
    skipnc
    goto       loop                ;no - read again
    movf       BANKMASK(tmp),w     ;yes - record this new high value
    movwf      BANKMASK(max)
    goto       loop                ;read again

    END        resetVec
```

4.1 Assembler Macros

An assembler macro was defined and used in this chapter's example program.

Assembler macros perform a similar function to the preprocessor's `#define` directive, in that they define a single identifier to represent a sequence of code. When a large amount of code spread over multiple lines is to be represented, assembler macro definitions might be easier to read than preprocessor macros.

In this chapter's example, the macro is defined to represent just one instruction, and is as shown here.

```
skipnc  MACRO
    btfsc     CARRY
ENDM
```

This creates a macro that skips the following instruction based on the carry bit in the STATUS register. Bit access of SFRs is described later, in section 7.2. Defining And Using Bits.

This macro is used just the once in this example, repeated here:

```
    subwf      max^(tmp&0ff80h),w
    skipnc
    goto       loop
```

An assembler macro can have arguments, too, and there are several characters that have special meaning inside macro definitions. A full description of these are presented in the *MPLAB® XC8 PIC Assembler User's Guide*.

4.2 User-defined Psects for Mid-range and Baseline Devices

This chapter's example code places the entry-point code associated with Reset in a user-defined psect, as was done in the equivalent PIC18 example code shown in section 3. A Basic Example For PIC18 Devices. The definition for this psect, however, looks a little different to that used by the PIC18 code and is given by the line:

```
PSECT resetVec,class=CODE,delta=2
```

Notice that the `reloc=2` flag that was used in the PIC18 example is not necessary with psects that hold code for Mid-range or Baseline devices. As this flag has not been specified, the `reloc` value defaults to 1, which implies that this psect will not be word aligned. Instructions on Mid-range or Baseline devices can be located at any address, so no word alignment is necessary.

The new psect flag that has been used with `resetVec` is the `delta` flag. A `delta` value of 2 indicates that 2 bytes reside at each address in the memory space where this psect will be located. This agrees with the program memory implemented on Mid-range and Baseline devices, which is either 14 or 12 bits wide (respectively) and which requires 2 whole bytes to program. PIC18 devices, on the other hand, define 1 byte of memory at each address, hence the `delta` value associated with psects holding code for those devices should be set to 1 (the default value if no `delta` flag is used).

Any program memory psect holding executable code on a Mid-range or Baseline devices must set the `delta` flag to 2. Psects destined for data memory or psects used with other devices should omit the `delta` flag or explicitly set the flag value to 1.

4.3 Working with Data Banks

The Mid-range code in this chapter's example shows how to handle data memory banking. Check your device data sheet to ensure you understand how banking works on your chosen device.

The equivalent PIC18 example code, shown in section 3. A Basic Example For PIC18 Devices, largely avoided banked data memory complications by using the `movff` instruction, which can access the entire data memory without the need for the bank selection register to be set. The `movffl` instruction can also access the entire data memory on PIC18 devices that have an expanded amount of data RAM, such as the 18F47K42. Additionally, the `max` and `tmp` objects were placed into a psect that was destined for Access bank memory, which can also be accessed using file register instructions without using the bank selection register.

Mid-range and Baseline PIC devices do not implement the `movff` instruction, so all movement of data is performed by banked instructions. So too, all other instructions that access file registers (e.g. `clrf`, `addwf`) are banked. Baseline and Mid-range devices also have only very limited amounts of common memory, which can be accessed independently of the currently selected bank, so most programs on Baseline and Mid-range devices will need to consider the memory banks used by objects.

To illustrate how banked instructions should be used, the code in this Mid-range example links the `tmp` and `max` objects into banked rather than common memory, by placing them in the assembler-defined psect `udata_bank0`. This means that the code that accesses these objects must deal with bank selection and address masking.

Bank selection (typically performed using the `BANKSEL` directive) must always be handled by the addition of instructions into your program, but there are two methods of handling the masking of addresses used by instruction operands. One is to manually mask the address by using either a predefined macro or by using an expression. The other method is to use a linker option to have the linker automatically truncate addresses to suite the instruction. This essentially suppresses the linker fixup overflow error that would normally occur if the address was not masked. Both these methods are discussed below.

Attention: The actions of selecting the bank of an object before that object is accessed and masking the address of an object when it is used as the operand to a file register instruction are quite independent. Regardless of whether you chose to manually mask addresses (e.g. using the `BANKMASK()` macro) or have the linker automatically truncate addresses, you must always include the appropriate bank selection sequences before objects are accessed. This is typically performed using the `BANKSEL` directive.

Manual Address Masking

The example program in this chapter shows bank selection and the addresses being manually masked. The main part of the example program begins as follows.

```
    BANKSEL    max                 ;starting point
    clrf       BANKMASK(max)
    BANKSEL    ANSELC
    clrf       BANKMASK(ANSELC)    ;select digital input for port C
```

```
loop:
    BANKSEL    PORTC                ;read and store port value
    movf       BANKMASK(PORTC),w
    BANKSEL    tmp
    movwf      BANKMASK(tmp)
    subwf      max^(tmp&0ff80h),w  ;is this value larger than max?
  ...
```

The `BANKSEL max` directive has been used to select the bank of the object `max` before it is accessed by the `clrf` instruction following. Although you can manually write the instruction or instructions needed to set the bank selection bits for the device you are using, the `BANKSEL` directive is more portable and is easier to interpret. On devices other than PIC18s and Enhanced Mid-range devices, this directive can sometimes expand into more than one instruction, so you should never use it immediately after an instruction that can skip, such as the `btfsc` instruction.

The `clrf` instruction clears the object `max`. Here, the `BANKMASK()` preprocessor macro has been used with the instruction operand to remove the bank bits contained in the address of `max`. This directive ANDs the address with an appropriate mask.

If this bank information is not removed, the linker might issue a fixup overflow error, which occurs when the operand value determined by the linker is too large for the relevant field in the instruction opcode. For example, Mid-range file register instructions have a 7-bit wide field in their op-code that specifies the offset into the currently selected bank of the location they are to access.

In the last few instructions of the code snippet shown above, the programmer has assumed that `max` and `tmp` are defined in the same psect, and hence will be located in the same data bank. The example code selects the bank of `tmp`, writes the W register into that object using a `movwf` instruction, and then `max` is accessed by the `subwf` instruction. Based on the programmer's assumption, a `BANKSEL` directive to select the bank of `max` before the `subwf` instruction is redundant, and it has been omitted to reduce the size of the program.

While the above assumption is valid and the code will execute correctly, it could fail if the definitions for `max` and `tmp` were to change and the objects ended up in different banks. Such a failure would be difficult to track down. Fortunately, there are other ways to remove the bank information from an address that can be used to your advantage.

The last line of code in the above example:

```
subwf   max^(tmp&0ff80h),w
```

contains a check to ensure that the bank of `tmp` and `max` are the same. Rather than ANDing out the bank information in `max` by using the `BANKMASK()` macro, the operand expression XORs (`^` operator) the full address of `max` with the bank bits contained in the address of `tmp` (which is obtained by masking out only the bank offset from `tmp` using the AND operator, `&`). If the bank bits in the address of `max` and the bank bits in the address of `tmp` are the same, they will XOR to zero in this expression. If they are not the same, they will produce a non-zero component in the upper bits of the address and trigger a fixup overflow error from the linker. This is much more desirable than the code silently failing at runtime.

The operand expression to the `subwf` instruction checks to ensure that two objects are in the same bank, but you can also use this style of check to ensure that an object is in a particular bank. If, for example, the code required that `max` must be in bank 2, then using the address expression `(max ^ 100h)` on a Mid-range device would trigger a fixup error if that was not the case. Here the value 0x100 consists of the bit sequence that represents bank 2, with all seven bits of the bank offset (the least significant bits) zeroed.

The section relating to file register address masking in the *MPASM™ to MPLAB® XC8 PIC® Assembler Migration Guide* has more information on the format of addresses and the appropriate masks to use, should you decide to use the AND or XOR operators and you not use the `BANKMASK()` macro.

Automatic Address Masking

The other method of handling address masking is to have the linker automatic truncate the addresses for you. This is by far the easiest to use and does not clutter your assembly source with ancillary expressions in instruction operands. To enable this, use the driver option `-Wl,--fixupoverflow=`*`action`* option, with `ignore` or `warn` and/or `lstwarn` as the *`action`* argument. Such an option will have the linker either totally ignore fixup overflow errors or only issue a warning should it encounter such situations. The address that will be used by the instruction

is the full operand address masked to the exact width expected by the instruction. See the *MPLAB® XC8 PIC® Assembler User's Guide* for more information on this option.

The above code extract taken from this chapter's example has been repeated here, showing how only code associated with bank selection is required if you are using this linker option.

```
    BANKSEL    max                 ;starting point
    clrf       max
    BANKSEL    ANSELC
    clrf       ANSELC              ;select digital input for port C
loop:
    BANKSEL    PORTC               ;read and store port value
    movf       PORTC,w
    BANKSEL    tmp
    movwf      tmp
    subwf      tmp,w               ;is this value larger than max?
  ...
```

Notice that the `BANKMASK()` macros have not been used, nor any other expressions to remove the bank information from the instruction operands; however, the code will still build and execute correctly if these are present. Note that this option does not affect bank selection in any way. The `BANKSEL` directive (or equivalent instructions) must always be used to select the bank of a memory location, regardless of how you handle address masking.

Throughout this document, source code examples will show instructions where their operand addresses have been manually masked, but the macros and expressions that perform this task do not need to be used if you prefer to use the `-Wl,--fixupoverflow` option and have the linker truncate addresses for you. As noted earlier, bank selection directives or instructions must always be present in the source when required.

4.4 Building the Example

If the entire example source code for this chapter was saved in a plain text file, called `readPort.S`, for example, it could be built using the command below. Note that the file name uses an upper case `.S` extension so that the file is preprocessed before any other processing.

```
pic-as -mcpu=16f18446 -Wl,-presetVec=0h -Wa,-a -Wl,-Map=readPort.map readPort.S
```

The psect containing the reset code has been linked to address 0. The command also includes the options to generate a map (`readPort.map`). The option, `-Wa,-a`, requests that an assembly list file be produced, so you can explore the generated code. This option creates the list file with the same base name as the first source file present in the command line and with the extension, `.lst`. The above command will generate a list file called `readPort.lst`.

To have the linker to automatically truncate addresses when building, as discussed in 4.3. Working with Data Banks, use a command line similar to:

```
pic-as -mcpu=16f18446 -Wl,-presetVec=0h -Wa,-a -Wl,-Map=readPort.map -Wl,--
fixupoverflow=ignore readPort.S
```

where the linker has been instructed by the highlighted option to truncate values to fit the instruction operand width without warning. The assembler's default action if the `-Wl,--fixupoverflow` option is absent is to truncate values to fit the instruction operand width and insert a warning marker into the assembly list file where any overflow occurred. This action can be explicitly requested using the option `-Wl,--fixupoverflow=lstwarn`. This option can be added as **Custom linker options**, as described below, if you are using the MPLAB X IDE.

If you are building in the MPLAB X IDE, add additional linker options (such as those specified using the driver option `-Wl,`*`linkerOption`*) to the **Custom linker options** field in the **pic-as Linker** > **general** category in the Project Properties dialog. When adding options to this field, do not include the leading `-Wl,`, as this prefix is added in by the IDE. Specify additional options to the assembler (such as the `-Wa,`*`assemblerOption`*) in the **Additional options** field in the **pic-as Global Options** category (the leading `-Wa,` must be specified). See 3.7. Building the Example for screen captures showing where these additional options should be added.

5. Multiple Source Files, Paging and Linear Memory Example

In this PIC16F18446 example, the code reads a number of values from PORTC and stores these into the elements of an array accessed using linear memory addressing. Linear memory access is only implemented on Enhanced Mid-range devices, but other aspects of this code are relevant for all devices The example source code has been intentionally split into two files to illustrate how code can use routines and objects defined in other modules, and the consequences this has on how program memory pages must be handled.

Note: This code example performs manual masking of instruction operand addresses to avoid fixup overflow errors. You can alternatively have the linker automatically truncate operand values when building, as discussed in 4.3. Working with Data Banks, so that the `BANKMASK()` and `PAGEMASK()` macros used in this example are not required.

PIC16 Example - file_1.S

```
/*
 * Take NUM_TO_READ samples of PORTC, storing this into an array accessed
 * using linear memory. NUM_TO_READ must be defined as a macro on the command-line.
 */

PROCESSOR 16F18446

#include <xc.inc>

CONFIG "FEXTOSC = OFF"     // External Oscillator not enabled
CONFIG "RSTOSC = HFINT1"   // Power-up default value for COSC bits->HFINTOSC (1MHz)
CONFIG "CLKOUTEN = OFF"    // CLKOUT disabled; i/o or oscillator on OSC2
CONFIG "CSWEN = ON"        // Clock Switch Enable->Writing to NOSC and NDIV allowed
CONFIG "FCMEN = ON"        // Fail-Safe Clock Monitor Enable->FSCM timer enabled

CONFIG "MCLRE = ON"        // Master Clear Enable->MCLR pin is Master Clear function
CONFIG "PWRTS = OFF"       // Power-up Timer Enable bit->PWRT disabled
CONFIG "LPBOREN = OFF"     // Low-Power BOR enable bit->ULPBOR disabled
CONFIG "BOREN = ON"        // Brown-out reset enable->Enabled, SBOREN bit is ignored
CONFIG "BORV = LO"         // Brown-out Reset Voltage Selection->VBOR set to 2.45V
CONFIG "ZCDDIS = OFF"      // Zero-cross circuit disabled at POR
CONFIG "PPS1WAY = ON"      // PPSLOCK set/cleared only once
CONFIG "STVREN = ON"       // Stack Over/Underflow causes reset

CONFIG "WDTCPS = WDTCPS_31"    // WDT Period Divider 1:65536; software control
CONFIG "WDTE = OFF"        // WDT operating mode->WDT Disabled, SWDTEN is ignored
CONFIG "WDTCWS = WDTCWS_7" // WDT Window always open; software control
CONFIG "WDTCCS = SC"       // WDT input clock selector->Software Control

CONFIG "BBSIZE = BB512"    // Boot Block Size Selection->512 words boot block size
CONFIG "BBEN = OFF"        // Boot Block Enable bit->Boot Block disabled
CONFIG "SAFEN = OFF"       // SAF Enable bit->SAF disabled
CONFIG "WRTAPP = OFF"      // Application Block not write protected
CONFIG "WRTB = OFF"        // Boot Block not write protected
CONFIG "WRTC = OFF"        // Configuration Register not write protected
CONFIG "WRTD = OFF"        // Data EEPROM write protection->Not write protected
CONFIG "WRTSAF = OFF"      // Storage Area Flash not write protected
CONFIG "LVP = ON"          // Low Voltage Programming Enabled, MCLR/Vpp pin is MCLR

CONFIG "CP = OFF"          // UserNVM Program memory code protection disabled

PSECT code
;read PORTC, storing the result into WREG
readPort:
    BANKSEL   PORTC
    movf      BANKMASK(PORTC),w
    return

GLOBAL count                       ;make this globally accessible
```

```
PSECT udata_shr
count:
    DS        1                    ;1 byte in common memory

PSECT resetVec,class=CODE,delta=2
resetVec:
    PAGESEL   main
    goto      main

GLOBAL storeLevel                  ;link in with global symbol defined elsewhere

PSECT code
main:
    BANKSEL   ANSELC
    clrf      BANKMASK(ANSELC)
    clrf      count
loop:
    ;a call to a routine in the same psect
    call      readPort             ;value returned in WREG
    ;a call to a routine in a different module
    PAGESEL   storeLevel
    call      storeLevel           ;expects argument in WREG
    PAGESEL   $
    ;wait for a few cycles
    movlw     0xFF
delay:
    decfsz    WREG,f
    goto      delay
    ;increment the array index, count, and stop iterating
    ;when the final element is reached
    movlw     NUM_TO_READ
    incf      count,f
    xorwf     count,w
    btfss     ZERO
    goto      loop

    goto      $                    ;loop forever

    END       resetVec
```

PIC16 Example - file_2.S

```
PROCESSOR 16F18446

#include <xc.inc>

GLOBAL storeLevel                  ;make this globally accessible
GLOBAL count                       ;link in with global symbol defined elsewhere

PSECT udata_shr
tmp:
    DS     1

;define NUM_TO_READ bytes of linear memory, at banked address 0x120
DLABS  1,0x120,NUM_TO_READ,levels

PSECT code
;store byte passed via WREG into the count-th element of the
;linear memory array, levels
storeLevel:
    movwf     tmp                  ;store the parameter
    movf      count,w              ;add the count index to...
    addlw     low(levels)          ;the base address of the aray...
    movwf     FSR1L                ;storing the result in FSR1
    movlw     high(levels)
    clrf      FSR1H
    addwfc    FSR1H
```

```
    movf      tmp,w                ;retrieve the parameter
    movwf     INDF1                ;access levels in linear memory
    return

    END
```

5.1 Multiple Source Files and Shared Access

Your assembly program can be split into as many source files as required. Doing so will make the source code easier to manage and navigate, but there are steps you must take to share objects and routines between modules, and your code will need to take into account where routines in other modules might ultimately be located.

To access an object or routine define in another source file, there are two steps that need to be followed. First, when you define the object or routine, you must tell the linker to allow the symbol (label) to be globally accessible. This is done using the `GLOBAL` directive. Next, you must declare the same symbol in every other file that needs to access it to inform the linker that this symbol is linked to a definition in another module. This can be done using either the `GLOBAL` or `EXTRN` directive.

In the full example program shown in this chapter, `file_1.S` contains the definition for the object `count`, which is essentially a label associated with reserved storage space, as shown below.

```
GLOBAL count

PSECT  udata_shr
count:
    DS    1                    ;1 byte in common memory
```

As `count` must be accessible from other modules, a `GLOBAL count` directive was used in addition to the symbol's definition, as shown above, to tell the linker that declarations of the same symbol in other modules can link to the definition in this module.

In `file_2.S`, another `GLOBAL count` directive was used to declare the symbol and link this with the definition of `count` that the linker will ultimately find in the `file_1.o` module. You may prefer to use the `EXTRN count` directive to perform this function. This directive performs a similar task to `GLOBAL`, but it will trigger an error if the directive appears in the same module that contains the symbol's definition.

The same mechanism is used for symbols used by routines that need to be accessible from more than one module. In this example, the code in `file_2.S` contains a `GLOBAL storeLevel` directive to allow the symbol to be referenced from outside that module, and `file_1.S` contains the same directive to link the `storelevel` symbol in that module with its definition in the other. Again, an `EXTRN storeLevel` directive could have been used in `file_1.S` to ensure the symbol is defined in the module you expect.

5.2 Psect Concatenation And Paging

All executable code must be placed in a psect. The grouping of psects is based on how and where the psects are defined, and this may affect how you write code that jumps to labels or calls other routines.

Notice that in `file_1.S` there were two separate blocks of assembly code placed into the `code` psect (one defining `readPort`, the other containing `main`). The content of psects with the same name are concatenated by the linker prior to memory allocation (unless they use the `ovrld` flag), thus the instruction associated with the label `main:` will occur directly after the last instruction in the `readPort` routine, even though there is other code and objects defined between these blocks in the source file.

Note that the content of the `code` psect in `file_2.S`, however, will not concatenate with the already combined content of the `code` psect in `file_1.S`. Concatenation of psects only occurs for psects with the same name and defined within the same module. In this case, the `code` psect in `file_2.S` will be linked independently to the `code` psect in `file_1.S`.

After building the example program, the map file will show two `code` psects linked at different addresses. The `code` psects are listed once in the module-by-module listing under `file_1.o` (the object file produced from `file_1.S`) and again under `file_2.o` in the map file extract below. They are also shown in the listing by class, but you

cannot see the source file in which they were defined there. Note that there is just the one `code` psect indicated for `file_1.o`, as both sections of code placed in that psect were concatenated.

```
                 Name                 Link     Load   Length Selector   Space Scale
file_1.o         resetVec                0        0        2        0       0
                 code                 3E7      3E7       19      7CE       0
                 udata_shr             71       71        1       70       1
file_2.o         code                 3DD      3DD        A      7BA       0
                 udata_shr             70       70        1       70       1

TOTAL            Name                 Link     Load   Length     Space
         CLASS   CODE
                 resetVec                0        0        2         0
                 code                 3E7      3E7       19         0
                 code                 3DD      3DD        A         0
```

The program memory on Baseline and Mid-range devices is paged, and your device data sheet will indicate the page arrangements for your device. The way psects are concatenated has consequences for how code written for these devices must call or jump to labels. The program memory on PIC18 devices is not paged. All addresses in their program memory are reachable by call and jump instructions, so the following discussion does not apply to programs written for these devices.

On Baseline and Mid-range devices, there are three methods of handling flow-control instructions. One is to manually add page selection instructions or directives into your program, then mask the address by using either a predefined macro or by using an expression. Another method is to again manually insert page selection sequences, but to use a linker option to have the linker automatically truncate addresses to suite the instruction. This essentially suppresses the linker fixup overflow error that would normally occur if the address was not masked. The final method is to use psuedo instructions that handle both page selection and address masking for you. All these methods are discussed below.

Manual Address Masking

Before making a call or jump on Baseline and Mid-range devices, the PCLATH register must contain the value (the upper bits of the address) to select the page of the destination. The `PAGESEL` directive can be used to initialize the PCLATH register for you.

You can see a `PAGESEL` directive being used before the `call storeLevel` instruction in the example code, repeated here:

```
loop:
    ;a call to a routine in the same psect
    call      readPort               ;value returned in WREG
    ;a call to a routine in a different module
    PAGESEL   storeLevel
    call      PAGEMASK(storeLevel)    ;expects argument in WREG
    PAGESEL   $
    ;wait for a few cycles
    movlw     0xFF
delay:
    decfsz    WREG,f
    goto      delay
```

Note that it is used again after the call using the current location counter, `$`, as its argument to ensure that PCLATH is again pointing to the page holding the code currently being executed. This allows the `goto delay` instructions following the call to work as expected.

As shown above, a `PAGESEL` directive was *not* used before the call to `readPort`. The PCLATH register did not need to be updated in this case because of two conditions. First, as mentioned earlier, the `code` psect that contains the `readPort` routine will concatenate with the `code` psect that holds the `main` routine, so these two routines (the caller and callee) will be in the same concatenated psect; and second, the `CODE` linker class associated with the `code` psect is defined in such a way that psects placed in its memory ranges can never cross a page boundary.

The linker options used when you build are shown in the map file. For the 16F18446 device used in this example, the top of the map file is as follows:

```
Linker command line:

-W-3 --edf=/Applications/microchip/xc8/v2.31/pic/dat/en_msgs.txt -cs \
  -h+dist/default/debug/asm_curiosity_16F18446_linearMemory.X.debug.sym \
  --cmf=dist/default/debug/asm_curiosity_16F18446_linearMemory.X.debug.cmf \
  -z -Q16F18446 -o/tmp/xckFLd4UW -presetVec=0h -ver=XC8 PIC(R) Assembler \
  -Mfile.map -E1 --acfsm=1493 -ACODE=00h-07FFhx8 -ASTRCODE=00h-03FFFh \  ...
```

The `CODE` class is defined by the linker option: `-ACODE=00h-07FFhx8`. This option indicates that the memory associated with the `CODE` class consists of 8 consecutive pages, the first starting at address 0, and each being 0x800 words long. These ranges correspond to the page addresses on the 16F18446 device.

The linker will never allow a psect placed into the memory associated with a class to cross boundaries in that class's memory ranges, which implies in this example that a psect placed in the `CODE` class will be wholly contained in a device page. You will receive a 'can't find space' error if a psect linked into this class exceeds the size of a page. If the class had instead been defined using `-ACODE=0-01FFFh`, it would cover exactly the same memory, but the boundaries in that memory would not exist. Psects placed in a class such as this could be linked anywhere, potentially straddling a device page boundary.

It is common to restrict where the linker can place psects so that assumptions can then be made in the source code that improve efficiency. In this case, the page boundaries in the `CODE` class, has meant that calls or jumps to a label that is defined in the same psect and in the same module do not need to first select the destination page (assuming that PCLATH already points to that page).

Page selection must be considered for any Baseline and Mid-range non-relative control instruction, those being the `goto`, `call`, and `callw` instructions. It is not needed prior to relative branch instructions; however, as these instructions modify PC once the branch has been taken, you will need to assess whether page selection is required for calls and jumps made after the branch. Page selection may also be required should you use any instructions that specify the PCL register as the destination, as these also use the PCLATH register to form the destination address.

A `PAGEMASK()` preprocessor macro has been used with the `call` instruction operand to remove the page bits contained in the address of `storelevel`. This directive ANDs the address with an appropriate mask. If this page information is not removed, the linker might issue a fixup overflow error, which occurs when the operand value determined by the linker is too large for the relevant field in the instruction opcode. For example, Mid-range `call` instructions have an 11-bit wide field in their op-code that specifies the offset into the currently selected page of the location they are to call.

Automatic Address Masking

The other method of handling address masking is to have the linker automatic truncate the addresses for you. This is by far the easiest to use and does not clutter your assembly source with ancillary expressions in instruction operands.

As with manual masking, before making a call or jump on Baseline and Mid-range devices, the PCLATH register must contain the value (the upper bits of the address) to select the page of the destination. The `PAGESEL` directive can be used to initialize the PCLATH register for you.

To enable automatic masking, use the driver option `-Wl,--fixupoverflow=`*`action`* option, with `ignore` or `warn` and/or `lstwarn` as the *`action`* argument. Such an option will have the linker either totally ignore fixup overflow errors or only issue a warning should it encounter such situations. The address that will be used by the instruction is the full opernad address masked to the exact width expected by the instruction. See the *MPLAB® XC8 PIC® Assembler User's Guide* for more information on this option.

The above code extract taken from this chapter's example has been repeated here, showing how it can be written if you are using this linker option.

```
loop:
    ;a call to a routine in the same psect
    call      readPort                ;value returned in WREG
    ;a call to a routine in a different module
    PAGESEL   storeLevel
    call      storeLevel    ;expects argument in WREG
    PAGESEL   $
```

```
    ;wait for a few cycles
    movlw     0xFF
delay:
    decfsz    WREG,f
    goto      delay
```

Notice that the `PAGEMASK()` macro has not been used; however, the code will still build and execute correctly if these are present. Note that this option does not affect page selection in any way. Instructions or the `PAGESEL` directive must always be used to select the page of a memory location, regardless of how you handle address masking.

Using Psuedo Instructions

The third way to handle flow-control is by use of two pseudo instructions. These both select the page of the destination and perform masking of the called address.

The psuedo instructions are `ljmp` and `fcall`. These expand to a `goto` and `call`, respectively, with the necessary page selection before the `goto` or `call` instruction, then page selection of the current page after the instruction. As the `ljmp` and `fcall` mnemonics can expand to more than one PIC instruction, they should never be used immediately after any instruction that skips, such as the `btfsc` instruction. You can see the opcodes generated for the `ljmp` or `fcall` pseudo instructions in the assembly list file.

The above code snippet could be rewritten:

```
loop:
    ;a call to a routine in the same psect
    call      readPort             ;value returned in WREG
    ;a call to a routine in a different module
    fcall      storeLevel          ;expects argument in WREG
    ;wait for a few cycles
    movlw     0xFF
delay:
    decfsz    WREG,f
    goto      delay
```

If you are not sure whether page selection is required before a call or jump, the safest approach is to use the `PAGESEL` directive or use the `fcall` and `ljmp` instructions, but just remember that doing so might unnecessarily increase the size of your code and slow its execution.

Paging Considerations

The above considerations might tempt you to place most of your code in the same psect in the one module so that you can avoid page selection instructions, but remember that once a psect grows larger than the size of a page, it will no longer fit in the `CODE` class and you'll receive 'can't find space' error message from the linker. Consider having frequently called routines and the routines that call them in psects with the same name and in the same module. Move other routines to other modules, or place them in psects with different names so that they are linked separately and can fill other device pages.

If you decide to create your own psects and linker classes to position code, keep in mind that how you define the linker classes might affect how your code needs to be written. If routines can straddle a bank boundary, they are more likely to fit in the program memory, but your program will require more page selection instructions. If you manually position psects to control where page selection instructions are needed (rather than have them linked anywhere in a linker class), this will require a lot of maintenance as the code is debugged and developed.

5.3 Linear Memory

The linear addressing mode is a means of accessing the banked data memory on Enhanced Mid-range devices as one contiguous and linear block. Your device data sheet will indicate if this memory is implemented on your device and contain further operational details.

Linear memory is typically used for objects that are too large to fit into a single memory bank, but it is important to remember that linear memory is not present in addition to a device's regular banked memory; it is simply an alternate

way to access that banked memory. You have to define objects larger than a bank in a different way to ordinary objects that are placed into banked memory.

This chapter's example code defines an object that is larger than a bank using the DLABS directive, shown here.

```
;define NUM_TO_READ bytes of linear memory, at banked address 0x120
DLABS  1,0x120,NUM_TO_READ,levels
```

This directive takes several arguments. The first is the address space in which the memory will be defined. This should be the value 1 for objects to be placed in data memory. The second argument is the starting address for the storage. This can be specified as either a linear or banked address. The equivalent linear address for the banked address 0x120, as used above, is 0x20A0, for example. The following argument is the number of bytes that is to be reserved. In this case, a preprocessor macro has been used to represent the array size. This macro has been defined in the build options, as shown in section 5.4. Building the Example. The final argument is a symbol. This is optional and creates a label for the object at the indicated address.

Unlike most directives, the DLABS directive does not produce output that is part of the current psect, so it can appear at any location in the source file. This also means that there should not be a label placed immediately before the directive. If a label is required for the object you are defining, then specify it as the last argument of the directive, as shown above.

In this example, the compiler will reserve storage for the levels object in banks 2 and 3, as specified by the address used with the directive. While it is possible to access this memory using the ordinary file register instructions, such access can become problematic if the offset into the array is not known, as it is not clear which bank needs to be selected beforehand. The Enhanced Mid-range devices, however, allow you to use the FSR register to indirectly read and write data memory using a linear address and without banking. Accessing levels indirectly is shown in the example and is repeated here.

```
storeLevel:
    movwf     tmp                 ;store the parameter
    movf      count,w             ;add the count index to...
    addlw     low(levels)         ;the linear base address of the array...
    movwf     FSR1L               ;storing the result in FSR1
    movlw     high(levels)
    clrf      FSR1H
    addwfc    FSR1H
    movf      tmp,w               ;retrieve the parameter
    movwf     INDF1               ;access levels using linear memory
```

Since it was defined using the DLABS directive, the symbol levels will always represent a linear address. This is still true even if a banked address was used as the second argument to this directive. Such an address can be loaded directly into an FSR register when linear access is required. The linear address value of the symbol can be found in the assembly list file and also in the map file if the symbol was made globally accessible using the GLOBAL directive.

5.4 Building the Example

If the two source code examples shown in this chapter were saved in plain text files called file_1.S and file_2.S, they could both be built using the single command below.

```
pic-as -mcpu=16f18446 -Wl,-presetVec=0h -DNUM_TO_READ=100 -Wa,-a -Wl,-
Map=linearMemory.map file_1.S file_2.S
```

In this command, the psect containing the reset code has been linked to address 0. The command also includes the option to generate a map file called linearMemory.map. The -Wa,-a option requests an assembly list file. One list file will be produced for each source module and the above command will generate files called file_1.lst and file_2.lst.

The preprocessor macro NUM_TO_READ, which determines the size of the linear array and the number of values stored, has been defined as 100 using the -D option. Defining this on the command line rather than using #define directives means that the macro can be more easily used in every source file. It also lets you change the value associated with the macro without having to modify your source code.

If you prefer to build each file separately (for example, from a make file) then you could also use the commands below:

```
pic-as -mcpu=16f18446 -Wa,-a -c -DNUM_TO_READ=100 file_1.S
pic-as -mcpu=16f18446 -Wa,-a -c -DNUM_TO_READ=100 file_2.S
pic-as -mcpu=16f18446 -Wl,-presetVec=0h -Wl,-Map=linearMemory.map file_1.o file_2.o
```

Here, the `-c` option has been used to generate an intermediate file for each source file. The intermediate (object) files will have a `.o` extension. The final command links the object files and produces the final output. Note that the option to create the assembly list file is required with the first two assembly commands, as is the `-D` option that defines the preprocessor macro. The option to create the map files is only needed with the final build command. The `-mcpu` option, which indicates the target device, must be used with every command.

The MPLAB X IDE performs incremental builds of each source file, in a similar way to the multi-step command-line example, above.

If you are building in the MPLAB X IDE, add additional linker options (such as those specified using the driver option `-Wl,`*`linkerOption`*) to the **Custom linker options** field in the **pic-as Linker** > **general** category in the Project Properties dialog. When adding options to this field, do not include the leading `-Wl,`, as this prefix is added in by the IDE. Specify additional options to the assembler (such as the `-Wa,`*`assemblerOption`*) in the **Additional options** field in the **pic-as Global Options** category (the leading `-Wa,` must be specified). See 3.7. Building the Example for screen captures showing where these additional options should be added.

To specify the `-D` option, enter `NUM_TO_READ=100` in the **Define preprocessor symbol** field in the **PIC-AS assembler** > **Preprocessing and Messages** category.

6. Compiled Stack Example

The example code in this chapter shows how you can use the linker to place objects on a compiled stack.

Not to be confused with a software stack, which is a dynamic stack allocation accessed via a stack pointer, a compiled stack is a memory area that is designated for static allocation of local objects that should only consume memory for the duration of a routine, in the same way that a function's auto and parameter objects behave in C programs.

The main advantage of using objects on a compiled stack is that the memory used by each routine for its local objects can potentially be reused by local objects defined by other routines. This substantially reduces the amount of data memory used by your program.

The drawback of a compiled stack is that the programmer must maintain the directives used to indicate the memory requirements of each routine and how those routines interact. Failure to do so could lead to code failure.

Compiled stacks do not use a stack pointer. Objects on such a stack are assigned a static address, which can be accessed via a symbol, thus there is no code size or speed penalty for using a compiled stack over ordinary objects. The allocation of memory on the stack is performed by the linker, so the total size of the stack can be determined, and the linker will issue a memory error if space cannot be found for the requested stack objects.

As the memory assigned to compiled stack objects is statically allocated, routines that use these objects are not reentrant, so they cannot be called recursively, nor can they be called from both main-line code and interrupt routines (or anything called by an interrupt routine).

Assembly code which illustrates the basic principles of using a compiled stack on a PIC18F47K42 device is shown below.

A Compiled Stack Example

```
#include <xc.inc>

CONFIG "FEXTOSC = OFF"          // External Oscillator not enabled
CONFIG "RSTOSC = HFINTOSC_1MHZ" // HFINTOSC, HFFRQ=4MHz, CDIV=4:1
CONFIG "CLKOUTEN = OFF"         // Clock out Enable->CLKOUT function is disabled
CONFIG "PR1WAY = ON"            // PRLOCKED One-Way cleared/set only once
CONFIG "CSWEN = ON"             // Writing to NOSC and NDIV is allowed
CONFIG "FCMEN = ON"             // Clock Monitor enabled

CONFIG "MCLRE = EXTMCLR"        // LVP=0: MCLR pin is MCLR; LVP=1: RE3 pin is MCLR
CONFIG "PWRTS = PWRT_OFF"       // PWRT is disabled
CONFIG "MVECEN = OFF"           // Vector table isn't used to prioritize interrupts
CONFIG "IVT1WAY = ON"           // IVTLOCK bit can be cleared and set only once
CONFIG "LPBOREN = OFF"          // ULPBOR disabled
CONFIG "BOREN = SBORDIS"        // Brown-out Reset enabled, SBOREN bit is ignored
CONFIG "BORV = VBOR_2P45"       // Brown-out Reset Voltage (VBOR) set to 2.45V
CONFIG "ZCD = OFF"              // ZCD disabled, enable by setting ZCDSEN in ZCDCON
CONFIG "PPS1WAY = ON"           // PPSLOCK cleared/set only once
CONFIG "STVREN = ON"            // Stack full/underflow will cause Reset
CONFIG "DEBUG = OFF"            // Background debugger disabled
CONFIG "XINST = OFF"            // Extended Instruction Set disabled

CONFIG "WDTCPS = WDTCPS_31"     // Divider ratio 1:65536; software control of WDTPS
CONFIG "WDTE = OFF"             // WDT Disabled; SWDTEN is ignored
CONFIG "WDTCWS = WDTCWS_7"      // window open 100%; software control
CONFIG "WDTCCS = SC"            // Software Control

CONFIG "BBSIZE = BBSIZE_512"    // Boot Block size is 512 words
CONFIG "BBEN = OFF"             // Boot block disabled
CONFIG "SAFEN = OFF"            // SAF disabled
CONFIG "WRTAPP = OFF"           // Application Block not incr protected
CONFIG "WRTB = OFF"             // Configuration registers not incr-protected
CONFIG "WRTC = OFF"             // Boot Block (000000-0007FFh) not incr-protected
CONFIG "WRTD = OFF"             // Data EEPROM not incr-protected
CONFIG "WRTSAF = OFF"           // SAF not Write Protected
```

```
CONFIG "LVP = ON"               // LV programming enabled, MCLR pin, MCLRE ignored

CONFIG "CP = OFF"               // PFM and Data EEPROM code protection disabled

;place the compiled stack in Access bank memory (udata_acs psect)
;use the ?au_  prefix for autos, the ?pa_  prefix for parameters
FNCONF udata_acs,?au_,?pa_

PSECT resetVec,class=CODE,reloc=2
resetVec:
    goto       main

PSECT code
;add needs 4 bytes of parameters, but no autos
FNSIZE add,0,4       ;two 2-byte parameters
GLOBAL ?pa_add

;add the two 'int' parameters, returning the result in
;the first parameter location
add:
    movf       ?pa_add+2,w,c
    addwf      ?pa_add+0,f,c
    movf       ?pa_add+3,w,c
    addwfc     ?pa_add+1,f,c
    return

;incr needs one 2-byte parameter
FNSIZE incr,0,2
GLOBAL ?pa_incr

;return the additional of the 2-byte parameter with the
;value in the W register
incr:
    addwf      ?pa_incr+0,c
    movlw      0h
    addwfc     ?pa_incr+1,c
    return

GLOBAL ?au_main
GLOBAL result
result  EQU    ?au_main+0          ;create an alias for this auto location

GLOBAL incval
incval EQU     ?au_main+2          ;create an alias for this auto location

FNROOT main                        ;this is the root of a call graph
FNSIZE main,4,0                    ;main needs two 2-byte 'autos' (for result and
incval)
FNCALL main,add                    ;main calls add
FNCALL main,incr                   ;main calls incr

PSECT code
main:
    clrf       result+0,c
    clrf       result+1,c
    movlw      2                       ;increment amount
    movwf      incval+0,c
    clrf       incval+1,c
loop:
    movff      result+0,?pa_add+0      ;load 1st parameter for add routine
    movff      result+1,?pa_add+1
    movff      incval+0,?pa_add+2      ;load 2nd parameter for add routine
    movff      incval+1,?pa_add+3
    call       add                     ;add result and incval
    movff      ?pa_add+0,result+0      ;store add's return value back to result
    movff      ?pa_add+1,result+1

    movff      incval+0,?pa_incr+0     ;load the parameter for incr routine
```

```
    movff       incval+1,?pa_incr+1
    movlw       2
    call        incr                    ;add 2 to incval
    movff       ?pa_incr+0,incval+0     ;store the result of incr back to incval
    movff       ?pa_incr+1,incval+1
    goto        loop

    END         resetVec
```

The function of the above assembly code is similar to that of the below C code.

```
int add(int a, int b) {
    return a + b;
}

void incr(int val, char amount) {
    return val + amount;
}

void main(void) {
    int result, incval;

    result = 0;
    incval = 2;
    while(1) {
        result = add(result, incval);
        incval = incr(incval, 2);
    }
}
```

6.1 Compiled Stack Directives

The assembler directives that control the compiled stack all begin with the letters `FN`.

The `FNCONF` directive is used once per program. It's three arguments indicate the name of the psect that should be used to hold the compiled stack, the symbol prefix to be used for auto-style objects, and the symbol prefix to be used for parameters objects.

In this example, the directive reads:

```
FNCONF udata_acs,?au_,?pa_
```

which indicates that the `udata_acs` psect (located in Access bank data memory) will be used to hold all the objects on the stack. The linker will form the stack in one contiguous chunk of memory. The location of the psect holding the stack will affect how the stack objects must be accessed. If there are a large number of stack-based objects and particularly if they are accessed often, placing the stack in the PIC18 Access bank will mean they can be accessed without any bank selection instructions. If the stack becomes too large, however, it will need to be placed in banked memory. Mid-range and Baseline devices have little common memory, and this might be needed for other purposes, so banked memory is often used for the compiled stack on these devices.

The `?au_` symbol specified as the second argument to the `FNCONF` directive will be prefixed to the name of a routine to create the base symbol for that routine's auto-style objects. The `?pa_` prefix will be used to form the base symbol for each routine's parameter objects. These symbols are illustrated in the `add` routine, which begins with the following code:

```
PSECT code
;add needs 4 bytes of parameters, but no autos
FNSIZE add,0,4       ;two 2-byte parameters
GLOBAL ?pa_add
;add the two 'int' parameters, returning the result in the first parameter location
add:
    movf        ?pa_add+2,w,c
    addwf       ?pa_add+0,f,c
```

The FNSIZE directive takes three arguments, those being the name of a routine, the total number of bytes required for that routine's auto-like objects, and the total number of bytes for its parameter-like objects. In this case, the FNSIZE directive indicates that the add routine needs no auto-style objects and 4 bytes of parameters. The directive can be placed anywhere in your code, but it is often located near the routine it configures.

The linker will automatically create a symbol associated with the block of 4 bytes used by the routine's parameters. In this case, that symbol will be ?pa_add, based on the prefix used with the FNCONF directive. Although this symbol is defined by the linker, it still needs to be declared in each module that needs it. This has been done in the above code by the GLOBAL ?pa_add directive. Each byte of the parameter memory can be accessed by using an offset from the ?pa_add symbol. The code above shows the first and third bytes of this memory being accessed. What these 4 bytes represent is entirely up to you. In the example, the parameter memory is used to hold two, 2-byte-wide objects, but the same FNSIZE arguments could instead be used to create one, 4-byte-wide object, for example.

Later in the example code, the following code begins the definition of the main routine.

```
GLOBAL ?au_main
GLOBAL result
result  EQU    ?au_main+0          ;create an alias for this auto location

GLOBAL incval
incval EQU     ?au_main+2          ;create an alias for this auto location

FNROOT main                        ;this is the root of a call graph
FNSIZE main,4,0                    ;main needs two 2-byte 'autos' (for result and
incval)
FNCALL main,add                    ;main calls add
FNCALL main,incr                   ;main calls incr

PSECT code
main:
    clrf       result+0,c
    clrf       result+1,c
```

The main routine requires four bytes of auto objects and so the FNSIZE directive has again been used to indicate this. The first byte of main's auto area can be accessed using the symbol ?au_main; however, in this instance, an EQU was defined so that the more readible name result could be used, as shown in the clrf instructions.

To be able to allocate memory on the stack, the linker needs to know how the program is structured in terms of calls. To allow it to form the program's call graph, several other directives are used.

The FNROOT directive, shown above, indicates that the specified routine forms the root node in a callgraph. (Alternatively, the resetVec label could have also been as the root node—it make no difference in this example.) The memory allocated to stack objects can be overlapped with that of other routines within the same callgraph, but no overlapping will take place between the stack objects of routines that are in different callgraphs. Typically you will define one callgraph root for the main part of your program and then one for each interrupt routine. This way, the stack memory associated with interrupt routines is kept separate and no data corruption can occur.

The FNCALL directive is used as many times as required to indicate which routines are called and from where. From this, the linker can form the callgraph nodes. In the above code sequence, the FNCALL directive was used twice. The first indicates that the main routine calls add; the second that main calls incr. As you develop your program, you need to ensure that there is an FNCALL directive for each unique call that takes place in the code. If the called routine does not define any compiled stack objects, the directive is not required, but it is good practice to include it anyway, in case there are changes made to the program.

You may devise whatever convention you like to pass arguments to routines. In the add routine, the LSB of the first parameter has an offset of 0; the MSB of the first parameter an offset of 1, etc., thus the expressions ?pa_add+0 and ?pa_add+1 represent the two bytes of the first 'int' parameter and ?pa_add+2 and ?pa_add+3 represent the two bytes of the second parameter. The first auto-style object defined by main is referenced using the expressions ?au_main+0 (or result+0) and ?au_main+1 (or result+1).

Typically, routines that need to return a value do so by storing that value into the memory taken up by their parameters. As the parameters should no longer be used once the routine returns, this reuse is not normally an issue, but you do need to consider the routine's return value when you allocate memory for the routine's parameters.

The parameter memory you request for a routine using the `FNSIZE` directive must be the larger of the total size of the routine's parameters and the size of the routine's return value.

6.2 Compiler Stack Allocation

When building your program, the linker processes all the FN-type directives in your program, generates the program's call graph, creates the symbols used by the stack objects (those that have `FNSIZE` directives), and allocates memory for these objects, overlapping them where possible. You can see the result of this process in the map file.

The call graph is printed towards the top of the map file, after the linker options and build parameters. For this example, it might look something like the following.

```
Call graph: (fully expanded)

*main size 0,4 offset 0
*    add size 4,0 offset 4
     incr size 2,0 offset 4
```

Indentation is used to indicate call depth. In the above, you can see that `main` calls `add` and `main` also calls `incr`. The size of the memory allocated for the routine's parameter and auto-style objects is printed, followed by the auto-parameter block (APB) offset into the compiled stack. Note that the offset is not an address; just a relative position in the stack.

Notice in the call graph that the offset of the APB for `add` and `incr` are identical. This implies that the memory they use is shared, which is possible because `add` and `incr` do not call each other in the code and so are never active at the same time.

A star, `*`, before a routine name indicates that the APB memory used by that routine is at a unique location and contributes to the total size of the program's RAM usage. These routines are critical path nodes in the call graph. Reducing the size of the APB used by these routines will reduce the total amount of data memory used by the program. The memory blocks used by unstarred routines totally overlap with blocks from other routines and do not contribute to the program's total data memory usage.

The `-mcallgraph` option can be used to customize what call graph information is displayed in the map file. Using `-mcallgraph=crit`, for example, will display only the nodes on critical paths, that is, all of the routines in the graph that are starred.

The advantage of using a compiled stack is obvious in this example: Although the program needed a total of 10 bytes of local storage, only 8 bytes needed to be allocated memory, with 2 bytes being reused. And this memory reduction was done without the programmer having to employ the dangerous practice of sharing objects between routines.

It is important to note that incorrect overlapping of APBs can occur if the information in the call graph is not accurate because of missing or erroneous FN-type directives in your program. Consider placing the `FNCALL` directive associated with a call immediately before the call instruction itself, so you can clearly see if one is missing. The FN-type directives can, however, be placed anywhere in the file, as they do not contribute to the hex file, and there is no harm in the same directive being repeated.

You will also see in the symbol table, printed at the end of the map file, the addresses assigned to the special linker-generated symbols used by the stack. For this program, it might look something like the following.

```
                                Symbol Table

?au_main          udata_acs     000000  ?pa_add           udata_acs     000004
?pa_incr          udata_acs     000004  __Hcode           code          000008
...
incval            udata_acs     000002  result            udata_acs     000000
```

Unlike the values printed for the offset in the call graph, the values in the symbol table are always absolute addresses. In this example, the `udata_acs` psect was linked to address 0, so the memory used by `main` for auto-style objects (`?au_main`) begins at address 0 and is shown to be in the `udata_acs` psect (as we specified). The memory used by `add` for its parameters (`?pa_add`) begins at address 4, as does the block used by `incr` (`?pa_incr`). The two equated symbols (`result` and `incval`) are also shown in this table.

6.3 Building the Example

If this chapter's example code was saved in a plain text file, called `cstack.S`, for example, it could be built using the command below. Note that the file name uses an upper case `.S` extension so that the file is preprocessed before any other processing.

```
pic-as -mcpu=18F47K42 -Wl,-presetVec=0h -Wl,-pudata_acs=COMRAM -Wa,-a -Wl,-
Map=cstack.map -mcallgraph=full cstack.S
```

No special options are needed for a compiled stack to be assembled by the linker; a compiled stack is automatically created if FN-type directives are detected in the object files being linked.

The above command shows a `-p` linker option being used to place the psect used to hold the compiled stack, `udata_acs`, into a suitable linker class. Such an option is not usually required, but will suppress a warning that will otherwise result when the `FNCONF` directive is used in a program.

The `-mcallgraph` option has been used to request that a full callgraph be printed. The callgraph is shown in the map file, which was requested by the option `-Wl,-Map=cstack.map` in the above.

As with the previous examples, the psect containing the reset code has been linked to address 0. The command also includes the `-Wa,-a` option to generate an assembly list file (`cstack.lst`), so you can explore the generated code.

If you are building in the MPLAB X IDE, add additional linker options (such as those specified using the driver option `-Wl,`*`linkerOption`*) to the **Custom linker options** field in the **pic-as Linker** > **general** category in the Project Properties dialog. When adding options to this field, do not include the leading `-Wl,`, as this prefix is added in by the IDE. Specify additional options to the assembler (such as the `-Wa,`*`assemblerOption`*) in the **Additional options** field in the **pic-as Global Options** category (the leading `-Wa,` must be specified). See 3.7. Building the Example for screen captures showing where these additional options should be added.

7. Interrupts and Bits Example For Mid-range Devices

The following example code is written for a PIC16F18446 and will execute on the Curiosity Nano board using that device. Much of this example is also applicable for other devices. The code initializes timer 0 to generate an interrupt every 500 mS, toggling the on-board LED (connected to port bit RA2) with each event.

Note: This code example performs manual masking of instruction operand addresses to avoid fixup overflow errors. You can alternatively have the linker automatically truncate operand values when building, as discussed in 4.3. Working with Data Banks, so that the `BANKMASK()` and `PAGEMASK()` macros used in this example are not required.

An Example of Interrupts and Bits

```
/*
 * Blink the LED on a PIC16F18446 Curiosity Nano Board
 * using the timer and interrupts to control the flash period.
 */

#include <xc.inc>

CONFIG "FEXTOSC = OFF"       // External Oscillator  not enabled
CONFIG "RSTOSC = HFINT1"     // Power-up default value for COSC->HFINTOSC (1MHz)
CONFIG "CLKOUTEN = OFF"      // Clock Out Enable->CLKOUT function disabled
CONFIG "CSWEN = ON"          // Clock Switch Enable->Writing to NOSC & NDIV allowed
CONFIG "FCMEN = ON"          // Fail-Safe Clock Monitor Enable->FSCM timer enabled

CONFIG "MCLRE = ON"          // Master Clear Enable->MCLR pin is Master Clear
CONFIG "PWRTS = OFF"         // Power-up Timer Enable bit->PWRT disabled
CONFIG "LPBOREN = OFF"       // Low-Power BOR enable bit->ULPBOR disabled
CONFIG "BOREN = ON"          // Brown-out reset enable->Enabled, SBOREN bit ignored
CONFIG "BORV = LO"           // Brown-out Reset Voltage Selection->VBOR set to 2.45V
CONFIG "ZCDDIS = OFF"        // Zero-cross detect disable->Disabled at POR
CONFIG "PPS1WAY = ON"        // PPSLOCK cleared/set only once
CONFIG "STVREN = ON"         // Stack Over/Underflow will cause a reset

CONFIG "WDTCPS = WDTCPS_31"  // WDT Period Select->Divider 1:65536; software control
CONFIG "WDTE = OFF"          // WDT operating mode->WDT Disabled, SWDTEN is ignored
CONFIG "WDTCWS = WDTCWS_7"   // WDT Window always open (100%); software control
CONFIG "WDTCCS = SC"         // WDT input clock selector->Software Control

CONFIG "BBSIZE = BB512"      // Boot Block Size->512 words boot block size
CONFIG "BBEN = OFF"          // Boot Block Enable bit->Boot Block disabled
CONFIG "SAFEN = OFF"         // SAF Enable bit->SAF disabled
CONFIG "WRTAPP = OFF"        // Application Block not write protected
CONFIG "WRTB = OFF"          // Boot Block Write not write protected
CONFIG "WRTC = OFF"          // Configuration Register not write protected
CONFIG "WRTD = OFF"          // Data EEPROM not write protected
CONFIG "WRTSAF = OFF"        // Storage Area Flash not write protected
CONFIG "LVP = ON"            // Low Voltage Programming enabled

CONFIG "CP = OFF"            // UserNVM Program memory code protection->Disabled

GLOBAL resetVec,isr
GLOBAL LEDState             ;make this global so it is watchable when debugging

PSECT bitbss,bit,class=BANK1,space=1
LEDState:
    DS          1           ;a single bit used to hold the required LED state

PSECT resetVec,class=CODE,delta=2
resetVec:
    ljmp        start

PSECT isrVec,class=CODE,delta=2
isr:
```

```
    ;no context save required in software for this device
    PAGESEL     $           ;select this page for the following goto
    BANKSEL     PIE0        ;for TMR0IE and TMR0IF
    ;for timer interrupts, set the required LED state
    btfsc       TMR0IE
    btfss       TMR0IF
    goto        notTimerInt ;not a timer interrupt
    bcf         TMR0IF
    ;toggle the desired bit state
    movlw       1 shl (LEDState&7)
    BANKSEL     LEDState/8
    xorwf       BANKMASK(LEDState/8),f
notTimerInt:
    ;code to handle other interrupts could be added here
exitISR:
    ;no context restore required in software
    retfie

PSECT code
start:
    ;set up the state of the oscillator and peripherals with RA2 as a digital
    ;output driving the LED, assuming that other registers have not changed
    ;from their reset state
    movlw       0x33
    BANKSEL     TRISA
    movwf       TRISA
    movlw       2
    BANKSEL     OSCFRQ
    movwf       OSCFRQ
    ;configure and start timer using interrupts
    movlw       0x89
    BANKSEL     T0CON1
    movwf       T0CON1
    movlw       0x1D
    movwf       TMR0H
    clrf        TMR0L
    BANKSEL     PIE0        ;for TMR0IE and TMR0IF
    bcf         TMR0IF
    bsf         TMR0IE
    movlw       0x80
    BANKSEL     T0CON0
    movwf       T0CON0
    bsf         GIE
    bsf         PEIE
loop:
    ;copy the desired state to the LED port pin
    BANKSEL     LEDState/8
    btfss       BANKMASK(LEDState/8),LEDState&7
    goto        lightLED
    BANKSEL     PORTA
    bsf         RA2         ;turn LED off
    goto        loop
lightLED:
    BANKSEL     PORTA
    bcf         RA2         ;turn LED on
    goto        loop

    END         resetVec
```

7.1 Interrupt Code (Mid-range)

The example shown in this chapter shows how to write an interrupt service routine (ISR).

An ISR is written in the same as any other routine, except it must:

- Have its entry point linked to the address of the relevant interrupt vector,

- Ensure that the content of any registers it uses and which are also used by main-line code are preserved,
- Ensure that only the relevant code is executed for each interrupt source, and
- Execute a special return-from-interrupt instruction to end the routine and interrupt processing.

These points are discussed below for the ISR in this example, which has been repeated below.

```
PSECT isrVec,class=CODE,delta=2
isr:
    ;no context save required in software for this device
    PAGESEL     $           ;select this page for the following goto
    BANKSEL     PIE0        ;for TMR0IE and TMR0IF
    ;for timer interrupts, set the required LED state
    btfsc       TMR0IE
    btfss       TMR0IF
    goto        notTimerInt ;not a timer interrupt
    bcf         TMR0IF
    ;toggle the desired bit state
    movlw       1 shl (LEDState&7)
    BANKSEL     LEDState/8
    xorwf       BANKMASK(LEDState/8),f
notTimerInt:
    ;code to handle other interrupts could be added here
exitISR:
    ;no context restore required in software
    retfie
```

The entry point to an ISR, like that of the code executed after Reset, must be linked to the appropriate interrupt vector address for the target device. To do this, the ISR, or at least the code containing the entry point to the ISR, should be placed in a unique psect that can be linked to the required address. In this example the `isrVec` psect holds the entire interrupt routine. Some PIC18 devices can employ an interrupt vector table and have different linking requirements for ISRs, as described in section 8. Interrupts and Bits Example For PIC18 Devices.

Any registers that are modified by an ISR (and any routines they call) and that are also used in main-line code must be saved on entry to the ISR and then restored on exit. For devices that support more than one interrupt vector (e.g. PIC18 devices), an ISR will need to save any registers that are modified but not saved by other ISRs. The code that saves and restores registers is often called context switch code.

Like many modern PIC devices, the 16F18446 automatically saves the state of core registers into shadow registers when an interrupt occurs, so context switching does not usually need to be performed in software for these devices, unless there are other registers or objects that the ISR should not leave modified. The example code above includes no context switch code at all, instead it immediately process the interrupt. If you need to write context switch code, see section 7.3. Manual Context Switch.

The ISR shown here toggles the desired state of the LED when a timer 0 interrupt occurs. It performs no action for any other interrupt source, but you could add code to this ISR to process as many interrupt sources as required. The `btfss TMR0IF` instruction checks the relevant timer interrupt flag to ensure that the timer 0 has triggered, but it does this in conjunction with the `btfsc TMR0IE` instruction, which checks the timer 0 interrupt enable bit. Only when both of these bits are set can you be sure that timer 0 is the source of the interrupt.

To terminate execution of the interrupt code and return to main-line code, use a `retfie` instruction, as shown. A regular `return` instruction will not return the state of the device to that when the interrupt occurred and will lead to code failure.

Since this ISR contains a `goto` instruction, a `PAGESEL` directive was added to ensure that the page that contains the ISR is selected and that the `goto` will reach the intended destination.

7.2 Defining And Using Bits

The interrupt routine shown in this chapter modifies a flag that is used by main-line code to set the state of the LED. The flag's state is set by the ISR. As this flag only needs to hold a true or false value, it was defined as a bit object to save on storage space and make checking its contents more efficient.

Bit objects are created in a similar way to ordinary objects, but a special flag must be used with the psect that holds them. The single bit-wide object called `LEDState` is defined by the following lines of code:

```
PSECT    bitbss,bit,class=BANK1,space=1
LEDState:
    DS          1           ;a single bit used to hold the required LED state
```

The inclusion of the `bit` flag with the psect definition instructs the linker that the units of address within this psect are bits, not bytes. This means that the `DS 1` directive, which allocates one unit of storage, is reserving a single bit, not a single byte. The `LEDState` object, then, can hold only a single bit.

The linker will allocate 8 bit objects to each byte of memory, so if there were additional allocations made to the above psect, like in the following:

```
PSECT    bitbss,bit,class=BANK1,space=1
LEDState:
    DS          1           ;a single bit used to hold the required LED state
otherState:
    DS          1           ;a single bit for some other purpose
```

then these two bits would reside in the same byte of memory, but would, of course, be at different bit positions within that byte.

Any symbol defined within a bit psect (for example `LEDState`) represents a bit address, not a byte address, and this affects how these symbols should be used in instructions. The PIC instructions that perform bit operations, such as `bcf` or `btfss`, require a byte address operand followed by a bit position within that byte. A bit object that is located at bit address 0x283, for example, is located at byte address `0x283/8`, which is 0x50, and at bit position `0x283&7`, which is position #3. If you are using a bit object with a bit instruction, then you will need to divide the bit address by 8 to obtain the byte (file register) address used in the instruction, and you will need to perform a bitwise AND with 7 to obtain the bit position within that byte. For example, the main-line example code that reads the desired state of the LED uses:

```
    BANKSEL     LEDState/8
    btfss       BANKMASK(LEDState/8),LEDState&7
```

Note that the `BANKSEL` directive also requires a byte address argument, so the bit address of `LEDState` was divided by 8 for this instruction as well. The `BANKMASK()` macro has also been used with the file register operand to the `btfss` instruction in the usual way, but make sure this macro is acting on the byte address of bit objects, that is, the bit address divided by 8.

You can, if desired, create a preprocessor macro to make the file address and bit position of an object more readable, for example

```
#define  LEDSTATE  BANKMASK(LEDState/8),LEDState&7
```

which could then be used as follows:

```
    BANKSEL     LEDState/8
    btfss       LEDSTATE
```

You will notice that bits within SFRs are predefined in the supplied header files in a similar way to `LEDSTATE`, above, and hence, they can be used in the same way. For example, the code sequence in the interrupt:

```
    btfsc       TMR0IE
    btfss       TMR0IF
    goto        notTimerInt ;not a timer interrupt
    bcf         TMR0IF
```

uses both the `TMR0IE` and `TMR0IF` SFR bits, both of which expand to the byte address that holds them (that being the addresses of PIE0 and PIR0, respectively) and their bit position within those bytes. These SFR bits are available once you include `<xc.inc>` into your source file.

The link address of any symbol defined in a bit psect is printed in the list and map files as a bit address. Do not compare such addresses to other byte addresses when checking memory allocation. You can confirm which psects

used the `bit` flag in the map file by looking for the **Scale** value. For bit psects, this will indicate a value of 8 and be left empty for non-bit psects. In the map file produced for this example project, the `bitbss` psect is shown.

```
      Name                           Link     Load   Length Selector   Space Scale
build/default/debug/main.o
      config                         8007     8007        5        0       4
      isrVec                            4        4        A        8       0
      resetVec                          0        0        3        0       0
      bitbss                          500       A0        1       A0       1     8
      code                              E        E       1D        8       0
```

Notice the link address of 0x500. This is a bit address. The load address, however, is converted to a byte address and is shown as 0xA0.

A bit psect can be linked anywhere in the data memory. To highlight how banking works with bit objects, the psect containing the flag in this example was associated with the `BANK1` linker class, which means that it was automatically placed somewhere in the bank 1 data memory. To access bit objects more efficiently, try to place them in the common memory or in the Access bank on PIC18 devices.

7.3 Manual Context Switch

If your device does not automatically save the state of core registers to shadow registers when an interrupt occurs, or if you need to preserve registers or objects in addition to those saved by hardware, the following steps need to be taken.

- Define sufficient storage in RAM for the registers that must be saved
- Have the ISR copy the contents of the relevant registers to the storage area before the registers are modified
- Have the ISR restore the contents of the registers from the storage area after those registers no longer need to be used

The following code snippet is written for a Mid-range device that does not automatically save context when an interrupt occurs, such as a PIC12F609. The code assumes the device implements common memory. This code illustrates the above points and is described in the following paragraphs. Adapt this code to suite your device and the registers you wish to preserve.

```
#include <xc.inc>

PSECT udata_shr
saved_WREG:
    DS      1               ;space for WREG in common memory

PSECT udata_bank0
mainSaveArea:               ;space for other registers in banked memory
saved_STATUS:
    DS      1
saved_PCLATH:
    DS      1

PSECT isrVec,class=CODE,delta=2
isr:
    ;save context
    movwf   saved_WREG      ;WREG saved to common memory
    movf    STATUS,w        ;STATUS (bank selection bits) copied to WREG...
    BANKSEL mainSaveArea    ;...allowing a different bank to be selected...
    movwf   saved_STATUS    ;STATUS saved
    movf    PCLATH,w
    movwf   saved_PCLATH    ;PCLATH saved

;interrupt code that can use STATUS, WREG, PCLATH goes here

exitISR:
    ;restore context
    BANKSEL mainSaveArea
    movf    saved_PCLATH,w
```

```
    movwf   PCLATH          ;PCLATH restored
    movf    saved_STATUS,w
    movwf   STATUS          ;STATUS restored
    swapf   saved_WREG,f
    swapf   saved_WREG,w    ;WREG restored without altering STATUS

    retfie
```

Space must be allocated for all registers that are to be preserved. This space does not need to be contiguous or even in the same bank. Storing context in common memory or in the PIC18 Access bank makes the context switching code much easier, but there might not be enough common memory for all the registers, nor might you wish to use this precious memory resource solely for interrupts.

In this example, one byte has been allocated (using the `DS` directive) in common memory (`udata_shr` psect) to hold the state of the W register, and memory was allocated in a block of bank 0 memory (`udata_bank0` psect) to store the state of the STATUS and PCLATH registers. This arrangement simplifies the context switch code, as you will see in the following discussion, but does not use much common memory. You may, as was done in this example, define a label with each byte of storage so that each byte can be uniquely referenced, or you may simply create one label and use an offset from that label to access each byte in that memory block. So, for example, `mainSaveArea + 1` and `saved_PCLATH` both represent the same memory location.

The code at the beginning of the ISR copies the registers to the allocated storage. You must ensure that this code does not overwrite any register that must be preserved until the content of that register has been saved. This might mean that the registers need to be saved in a particular order.

The first instruction in the above example moves the current content of the W register into the `saved_WREG` object. Because the storage for this object was allocated in common memory, the bank of this memory does not need to be selected. This is important because on Mid-range devices, the bank selection bits are in the STATUS register, which is one of the registers that must be preserved. A new bank cannot be selected without clobbering the STATUS register; similarly, the STATUS register cannot be saved first because that code would clobber the W register.

With the W register saved, the subsequent instruction can then load the W register with the content of the STATUS register. And with the STATUS register now copied (although not yet stored) the bank of the main storage block can be selected using the `BANKSEL` directive. The subsequent code can then use regular `movf`/`movwf` instructions to save the remaining registers.

The restoration code at the end of the ISR typically restores the registers in the reverse order to that in which they were saved. You must ensure that once a register has been restored, no subsequent code writes to that register, so again, these operations need to be performed in a strict order and may not be able to use some instructions.

The restoration code in this example first selects the bank of the main allocated space and copies the saved values back to the PCLATH and STATUS registers using `movf`/`movwf` instructions. Remember that with the STATUS register restored, the bank currently selected also returns to that in effect when the interrupt occurred. As the content of the remaining register to be restored in this example was stored in common memory, this is of no concern.

Restoring the W register is more difficult. The instruction, `movf saved_WREG,w`, can't be used because it affects the Zero bit in the STATUS register, which has already been restored. Instead, the example code restores the W register using two `swapf` instructions, which do not affect the STATUS register.

7.4 Building the Example

If this chapter's source code was saved in a plain text file, called `timedLED.S`, for example, it could be built using the command below. Note that the file name uses an upper case `.S` extension so that the file is preprocessed before any other processing.

```
pic-as -mcpu=16F18446 -Wl,-presetVec=0h -Wl,-pisrVec=04h -Wa,-a -Wl,-
Map=timedLED.map timedLED.S
```

The psect containing the reset code has been linked to address 0 and in addition, the psect holding the interrupt routine was linked to the device's vector location, address 0x4. The command also includes the `-Wl,-Map=timedLED.map` option to generate a map file and the `-Wa,-a` option to generate an assembly list file, which will be called `timedLED.lst`.

If you are building in the MPLAB X IDE, add additional linker options (such as those specified using the driver option `-Wl,`*`linkerOption`*) to the **Custom linker options** field in the **pic-as Linker** > **general** category in the Project Properties dialog. When adding options to this field, do not include the leading `-Wl,`, as this prefix is added in by the IDE. Specify additional options to the assembler (such as the `-Wa,`*`assemblerOption`*) in the **Additional options** field in the **pic-as Global Options** category (the leading `-Wa,` must be specified). See 3.7. Building the Example for screen captures showing where these additional options should be added.

8. Interrupts and Bits Example For PIC18 Devices

The following example code is written for a PIC18F47K42 and will execute on the Curiosity Nano board using that device. The code initializes timer 0 to generate an interrupt every 500 mS, toggling the on-board LED (connected to port bit RE0) with each event.

Note: This code example performs manual masking of instruction operand addresses to avoid fixup overflow errors. You can alternatively have the linker automatically truncate operand values when building, as discussed in 4.3. Working with Data Banks, so that the `BANKMASK()` and `PAGEMASK()` macros used in this example are not required.

An Example of Interrupts and Bits

```
/*
 * Blink the LED on a PIC18F47K42 Curiosity Nano Board
 * using the timer and interrupts to control the flash period.
 */

#include <xc.inc>

CONFIG "FEXTOSC = OFF"          // External Oscillator  not enabled
CONFIG "RSTOSC = HFINTOSC_1MHZ" // Reset Oscillator->HFINTOSC, HFFRQ=4MHz, CDIV=4:1
CONFIG "CLKOUTEN = OFF"         // Clock out Enable->CLKOUT function is disabled
CONFIG "PR1WAY = ON"            // PRLOCK cleared/set only once
CONFIG "CSWEN = ON"             // Writing to NOSC and NDIV is allowed
CONFIG "FCMEN = ON"             // Clock Monitor enabled

CONFIG "MCLRE = EXTMCLR"        // LVP=0=>MCLR pin is MCLR; LVP=1=>MCLR on RE3
CONFIG "PWRTS = PWRT_OFF"       // Power-up timer selection bits->PWRT is disabled
CONFIG "MVECEN = ON"            // Multi-vector enable->Multi-vector table enabled
CONFIG "IVT1WAY = ON"           // IVTLOCK cleared/set only once
CONFIG "LPBOREN = OFF"          // Low Power BOR Enable bit->ULPBOR disabled
CONFIG "BOREN = SBORDIS"        // BOR enabled, SBOREN ignored
CONFIG "BORV = VBOR_2P45"       // Brown-out Reset Voltage->VBOR set to 2.45V
CONFIG "ZCD = OFF"              // ZCD disabled; enable by setting ZCDSEN
CONFIG "PPS1WAY = ON"           // PPSLOCK cleared/set only once
CONFIG "STVREN = ON"            // Stack Full/underflow => Reset
CONFIG "DEBUG = OFF"            // Debugger Enable->Background debugger disabled
CONFIG "XINST = OFF"            // Extended Instruction Set Enable->Disabled

CONFIG "WDTCPS = WDTCPS_31"     // WDT Period->Divider 1:65536; software control
CONFIG "WDTE = OFF"             // WDT Disabled; SWDTEN is ignored
CONFIG "WDTCWS = WDTCWS_7"      // WDT window open (100%); software control
CONFIG "WDTCCS = SC"            // WDT input clock selector->Software Control

CONFIG "BBSIZE = BBSIZE_512"    // Boot Block Size->Boot Block size is 512 words
CONFIG "BBEN = OFF"             // Boot Block enable bit->Boot block disabled
CONFIG "SAFEN = OFF"            // Storage Area Flash enable bit->SAF disabled
CONFIG "WRTAPP = OFF"           // Application Block  not protected
CONFIG "WRTB = OFF"             // Configuration Register not protected
CONFIG "WRTC = OFF"             // Boot Block not write-protected
CONFIG "WRTD = OFF"             // Data EEPROM not write-protected
CONFIG "WRTSAF = OFF"           // SAF not Write Protected
CONFIG "LVP = ON"               // Low Voltage Programming Enable->LVP enabled

CONFIG "CP = OFF"               // PFM and Data EEPROM Code Protection->Disabled

GLOBAL resetVec
GLOBAL LEDState             ;make this global so it is watchable when debugging
GLOBAL __Livt               ;defined by the linker but used in this code

PSECT bitbssCOMMON,bit,class=COMRAM,space=1
LEDState:
    DS          1           ;a single bit used to hold the required LED state
```

```
PSECT resetVec,class=CODE,reloc=2
resetVec:
    goto        start

;vector table
PSECT ivt,class=CODE,reloc=2,ovrld
ivtbase:
    ORG         31*2           ;timer 0 vector position
    DW          tmr0Isr shr 2  ;timer 0 ISR address shifted right

PSECT textISR,class=CODE,reloc=4
tmr0Isr:
    bcf         TMR0IF      ;clear the timer interrupt flag
    ;toggle the desired LED state
    movlw       1 shl (LEDState&7)
    xorwf       LEDState/(0+8),c

    retfie      f

PSECT code
start:
    bsf         BANKMASK(INTCON0),INTCON0_IPEN_POSN,c        ;set IPEN bit
    ;use the unlock sequence to set the vector table position
    ;based on where the ivt psect is linked
    bcf         GIE
    movlw       0x55
    movwf       BANKMASK(IVTLOCK),c
    movlw       0xAA
    movwf       BANKMASK(IVTLOCK),c
    bcf         IVTLOCKED
    movlw       low highword __Livt
    movwf       BANKMASK(IVTBASEU),c
    movlw       high __Livt
    movwf       BANKMASK(IVTBASEH),c
    movlw       low __Livt
    movwf       BANKMASK(IVTBASEL),c
    movlw       0x55
    movwf       BANKMASK(IVTLOCK),c
    movlw       0xAA
    movwf       BANKMASK(IVTLOCK),c
    bsf         IVTLOCKED
    ;set up the state of the oscillator and peripherals with RE0 as a digital
    ;output driving the LED, assuming that other registers have not changed
    ;from their reset state
    movlw       6
    movwf       BANKMASK(TRISE),c
    movlw       0x62
    movlb       57
    movwf       BANKMASK(OSCCON1),b
    clrf        BANKMASK(OSCCON3),b
    clrf        BANKMASK(OSCEN),b
    movlw       2
    movwf       BANKMASK(OSCFRQ),b
    clrf        BANKMASK(OSCTUNE),b
    ;configure and start timer interrupts
    movlb       57
    bsf         TMR0IP
    movlw       0x6D
    movwf       BANKMASK(T0CON1),c
    movlw       0xF3
    movwf       BANKMASK(TMR0H),c
    clrf        BANKMASK(TMR0L),c
    movlb       57
    bcf         TMR0IF
    bsf         TMR0IE
    movlw       0x80
    movwf       BANKMASK(T0CON0),c
    bsf         GIEH
```

```
loop:
    ;set LED state to be that requested by the interrupt code
    btfss       LEDState/8,LEDState&7,c
    goto        lightLED
    bsf         RE0          ;turn LED off
    goto        loop
lightLED:
    bcf         RE0          ;turn LED on
    goto        loop

    END         resetVec
```

8.1 Interrupt Code (PIC18)

Much of the discussion in section 7.1. Interrupt Code (Mid-range) also applies to PIC18 interrupt code, and that information is not repeated here. In this section, PIC18-specific details concerning interrupts are discussed.

Some PIC18 devices, such as the PIC18F47K42 device used in this example, can be configured to use a table of interrupt vectors rather than two fixed interrupt vector locations. If your PIC18 device does not support vector tables or you are using it in legacy mode, you will need to define one or two interrupt service routines (one for each interrupt priority being used), whose entry points are linked to the interrupt vector addresses. This approach is similar to that shown in the Mid-range example; however, Mid-range devices only support one interrupt vector location.

The example in this chapter assumes the vectored interrupt controller is being used. Turning on the `MVECEN` configuration bit enables this mode of operation.

When using vector tables, one stand-alone interrupt service routine (ISR) is written for each interrupt source to be handled. A vector table containing the addresses of these ISRs is created and located in memory. The position of the vector within the table determines which source it handles. The base address of the table must be stored in the IVTBASE registers so the device will load the correct interrupt vector when an interrupt occurs. The content of the IVTBASE registers can be changed at runtime, meaning that a program can define more than one interrupt vector table, hence different ISRs can be selected at different points in the program.

The interrupt code in this example, and which is repeated here, shows one ISR (`tmr0Isr`) and its corresponding address in the vector table, whose base address is `ivtbase`.

```
;vector table
PSECT ivt,class=CODE,reloc=2,ovrld
ivtbase:
    ORG         31*2            ;timer 0 vector position
    DW          tmr0Isr shr 2   ;timer 0 ISR address shifted right

;one ISR to handle timer0
PSECT textISR,class=CODE,reloc=4
tmr0Isr:
    bcf         TMR0IF        ;clear the timer interrupt flag
    ;toggle the desired LED state
    movlw       1 shl (LEDState&7)
    xorwf       LEDState/(0+8),c

    retfie      f
```

The ISR to handle the timer 0 in this example uses the label `tmr0Isr`. Although the psect in which this symbol was placed can be linked anywhere in the program memory, the ISR entry point must be at an address that is a multiple of 4. This is due to the PIC18 device left shifting the value stored in the vector table by two bits to form the ISR address it will jump to. To ensure the ISR is located correctly, the psect it is placed in has a `reloc` value set to 4, which ensure it will begin on an address with a multiple of 4.

A `retfie f` instruction was used to return from the interrupt. Like many modern devices, the PIC18F47K42 automatically saves the state of core registers into shadow registers when an interrupt occurs, so context switching does not usually need to be performed in software, unless there are other registers or objects that the ISR should not leave modified. The example code above includes no context switch code at all, instead it immediately process

the interrupt. If you need to write context switch code, see section 7.3. Manual Context Switch. The `f` operand to the `retfie` instruction indicates the fast form of a return from interrupt, which will copy the content of the shadow registers back to the original registers.

The vector table was placed into a user-defined psect called `ivt`, which can then be linked independently to other psects in the program memory, as shown in section 8.3. Building the Example. The `reloc=2` flag was used to ensure that the psect's starting address is word aligned for correct operation, as described in the device data sheet. The use of the `ovrld` psect flag is explained later.

The assembler's `DW` directive was used to place the address of the ISR, `tmr0Isr`, into the vector table. The device expects this address to be shifted right by 2 bits, so the assembler's `shr` operator was used to obtain this value. The timer 0 interrupt vector must be located at position #31 in the table; however, the table itself can be moved around in program memory, so there is no absolute location for this (or any other) vector. To make it easier to modify the program, an `ORG` directive was used to locate the vector. This is one instance where the use of this directive is recommended. Note that the `ORG` directive moves the location counter *relative to the start of the current psect*, not to an absolute location, but that is exactly what is required in this situation. Provided the psect holding the vector table is linked to the required address, the `ORG` directive will ensure that the vector is at the correct offset within that the table. The vector is offset by the amount `31 * 2`, since each vector is two bytes wide.

As the program is developed and other interrupts are required, additional ISRs can be written and the addresses of these routines added to the vector table. The psect holding the vector table can be built up in sections and across multiple files. You might prefer to supply one ISR and its vector table contribution together in a source file of their own.

To ensure that the vectors in the psect appear in the correct order regardless of where and when each vector is added, the `ovrld` flag was used with the `ivt` psect. This flag tells the linker to overlay, rather than concatenate, each contribution to the psect. For example, the following additional code, which could be located in a different source file, places the vector for a UART1 receive interrupt routine.

```
PSECT ivt,class=CODE,reloc=2,ovrld
    ORG         27*2          ;UART1 receive vector position
    DW          uart1Isr shr 2

PSECT textISR,class=CODE,reloc=4
uart1Isr:
    ;code to process uart1 receive goes here
    retfie      f
```

Note that this vector is placed in the same psect that was used to hold the vector for the timer interrupt (`ivt`). Since the `ovrld` flag has been used with this psect, the `ORG` directive will again move the address that follows relative to the start address of the psect, even if the content of `ivt` psect from the first example has already been processed. In this way, you do not need to ensure that the code that defines the vectors are processed in any particular order.

If you are checking the vector table psect in the map file, you should see the Link address indicate the address at which it was placed (in the example below, address 8) and the length will include any gaps introduced by the `ORG` directive, or directives. Even though there is only one vector in the table in this example, the size has been shown as 0x40, due to the fact that the timer 0 vector is located at an offset of 31 * 2 from the beginning of the psect and is 2 bytes long.

```
TOTAL         Name                    Link     Load   Length     Space
  CLASS   CODE
              ivt                        8        8       40         0
```

Even if your project does not need interrupts from other sources, it is advisable to consider what will happen should any of these interrupt sources inadvertently trigger. You might want to populate the unused vector table locations with the address of a default ISR that can handle these cases.

The vector table can be positioned (and repositioned) in program memory by writing the appropriate table address to the IVTBASE registers. To avoid having to modify your source code should you decide to link the vector table to a different address, you can load the IVTBASE registers with special symbols that are defined by the linker. The

following extract from the example shows the IVTBASE unlock sequence (described in the device data sheet) and all three IVTBASE registers being loaded.

```
GLOBAL __Livt                ;defined by the linker but used in this code
...
    movlw       0x55
    movwf       BANKMASK(IVTLOCK),c
    movlw       0xAA
    movwf       BANKMASK(IVTLOCK),c
    bcf         IVTLOCKED
    movlw       low highword __Livt
    movwf       BANKMASK(IVTBASEU),c
    movlw       high __Livt
    movwf       BANKMASK(IVTBASEH),c
    movlw       low __Livt
    movwf       BANKMASK(IVTBASEL),c
```

For any psect allocated memory using a `-p` linker option, the linker defines special symbols that represent where in memory those psects were linked. The starting address of such psects are represented by a symbol with the form `__L`*psectname* (note the two leading underscore characters). (The linker also defines `__H`*psectname* and `__B`*psectname* symbols.) In this example, the `__Livt` symbol (the lower bound of the `ivt` psect) has been used to load the IVTBASE registers. The `low`, `high`, and `highword` operators have been used to obtain the appropriate byte of this full address for each of the registers. Such code does not need to be modified if at a later time you wish to change the location of the vector table; simply relink the program with a different `ivt` address.

8.2 Defining And Using Bits

The interrupt routine shown in this chapter modifies a flag that is used by main-line code to set the state of the LED. The flag's state is set by the ISR. As this flag only needs to hold a true or false value, it was defined as a bit object to save on storage space and make checking its contents more efficient.

Bit objects for PIC18 devices are created as per Mid-range devices (discussed in section 7.2. Defining And Using Bits), using the `bit` psect flag with the psect that holds them. The only difference to the following code extract compared to the that for the Mid-range device is that the psect was placed in the Access bank.

```
PSECT bitbssCOMMON,bit,class=COMRAM,space=1
LEDState:
    DS          1               ;a single bit used to hold the required LED state
```

Bit symbols are used in PIC18 code in the same way as for Mid-range devices. The file register address of a bit symbol can be obtained by dividing the bit address by 8, and the bit position can be obtained by taking the bitwise AND with 7. And as with Mid-range devices, once you include `<xc.inc>` into your program, bit symbols are defined for bits within special function registers, such as `TMR0IF` and `GIEH`, which were used in the example code.

You may notice in the example, repeated below, that the IPEN bit in the INTCON0 register was not set using an instruction like `bsf IPEN`, but rather, the operand to this instruction was specified as the INTCON0 file register address and a special symbol used to locate the IPEN bit within this register.

```
PSECT code
start:
    bsf         BANKMASK(INTCON0),INTCON0_IPEN_POSN,c        ;set IPEN bit
```

In this particular case, attempting to use the IPEN bit directly would result in an undefined symbol error. This is because the 18F47K42 device has more than one bit within its SFRs called IPEN. The other is the IPEN bit in the ADCON1 register. In such instances, the name IPEN is not unique and no bit symbol will be available in the provided headers. You must access the bit using the SFR name, but the headers do provide macros that represent the bit position, such as the `INTCON0_IPEN_POSN`, used above. The same headers also supply macros which indicate the number of bits in the SFR, e.g., `_INTCON0_IPEN_SIZE` and a mask you can use for bitwise logic operations, e.g., `_INTCON0_IPEN_MASK`. These macros are available for every SFR bit, and you can use them in preference to the bit name if you prefer.

8.3 Building the Example

If the source code shown in this chapter was saved in a plain text file, called `timedLED.S`, for example, it could be built using the command below. Note that the file name uses an upper case `.S` extension so that the file is preprocessed before any other processing.

```
pic-as -mcpu=18F47K42 -Wl,-presetVec=0h -Wl,-pivt=08h -Wa,-a -Wl,-Map=timedLED.map
timedLED.S
```

The `ivt` psect holding the vector table was linked to address 0x8. The link address of this psect is used in the source code to load the value contained in the IVTBASE register, to ensure that the device reads the correct vectors when an interrupt occurs. The address you specify for this psect must be a multiple of two to ensure the requirement imposed by the `reloc` flag used with this psect is met. If you were to attempt to link the psect at an odd address, an error will be issued.

The above command also includes the `-Wl,-Map=timedLED.map` option to generate a map file and the `-Wa,-a` option to generate an assembly list file, which will be called `timedLED.lst`.

If you are building in the MPLAB X IDE, add additional linker options (such as those specified using the driver option `-Wl,`*`linkerOption`*) to the **Custom linker options** field in the **pic-as Linker** > **general** category in the Project Properties dialog. When adding options to this field, do not include the leading `-Wl,`, as this prefix is added in by the IDE. Specify additional options to the assembler (such as the `-Wa,`*`assemblerOption`*) in the **Additional options** field in the **pic-as Global Options** category (the leading `-Wa,` must be specified). See 3.7. Building the Example for screen captures showing where these additional options should be added.

9. Baseline Code Example

Coding strategies for Baseline devices generally follow those for Mid-range devices, taking into account the smaller memory and the reduced feature set implemented by these devices. However, the paging issues associated with function calls on Baseline devices are different to other 8-bit PIC devices and need special attention.

The following Baseline code example writes increasing values to PORTA.

Baseline Code Example Showing Calls

```
#include <xc.inc>

PROCESSOR 16F570

; PIC16F570 Configuration Bit Settings
CONFIG  "FOSC = EXTRC_CLKOUT"  // Oscillator (EXTRC with CLKOUT on OSC2/CLKOUT)
CONFIG  "WDTE = OFF"           // Watchdog Timer Enable bit (Disabled)
CONFIG  "CP = OFF"             // Code Protection bit (Code protection off)
CONFIG  "IOSCFS = 8MHz"        // Internal Oscillator Frequency (8 MHz INTOSC Speed)
CONFIG  "CPSW = OFF"           // Flash Data Memory (Code protection off)
CONFIG  "BOREN = OFF"          // BOR Disabled
CONFIG  "DRTEN = OFF"          // DRT Disabled

PSECT udata_shr                // objects in common memory
counter:
    DS      1
loc:
    DS      1

PSECT  resetVec,class=CODE,delta=2
resetVec:
    ljmp    main

PSECT code
main:
    movlw   0
    tris    PORTA
    movf    counter,w
    /* write increasing values to PORTA */
loop:
    fcall   increment
    movwf   PORTA
    goto    loop

/* increment the value in W, returning it in W */
PSECT entryCode,class=ENTRY,delta=2
increment:
    movwf   loc
    incf    loc,w
    return

    END     resetVec
```

9.1 Routine Entry Points

The example code shown above looks similar to that appropriate for a Mid-range device, except for the psect used to hold the `increment` routine. This psect is associated with a special class that ensures it will be linked in such a way that calls to the label(s) at the start of the psect will work as expected on Baseline devices. This is explained below.

When a `call` instruction is executed by a PIC Baseline device, bits 0 through 7 of the PC register are loaded from the instruction operand, bits 9 and 10 are loaded from the PA bits in the STATUS register but, importantly, bit location 8 is unconditionally cleared. This implies that `call` instructions on Baseline devices can only reach addresses that fall in the first 256 locations of any program memory page. In other words, any entry points to callable routines must

be located in the first 256 program memory locations of a page. This is also true of any instruction that modifies the PCL register.

Note that this restriction does not affects jumps (`goto` instructions), which can reach any address. Also note that this restriction does not prevent a routine from consuming the entire program memory page; it merely means that the routine's *entry points* must be in the lower portion of the page if the routine is called or reached via an instruction that modifies the PCL register. Once code in a routine has been reached, sequential execution of the remainder of the code in that routine can proceed all the way to the end of the page.

To ensure that Baseline routines can be called without placing any further restrictions on the routines' length or location, linker classes can be defined using a modified syntax to specify an entry range. Normally, the option to define a linker class might look something like:

```
-AMYCLASS=00h-01FFh
```

which defines a class called `MYCLASS` with the address range 0 through to 0x1FF. If a routine was linked into this class, the linker could placed it anywhere in this range. But classes can be defined using an additional hyphen-separated address field, for example:

```
-AMYCLASS=00h-0FFh-01FFh
```

which tells the linker that when linking any psect into this range, the start of the psect must fall in first specified address range (0 through to 0xFF), but that the remainder of the psect can extend all the way up to the end address, in this case 0x1FF.

The PIC Assembler automatically defines such a linker class that can be used for linking Baseline psects that contain callable code entry points. The class is called `ENTRY` and you can see this class being used for the psect that holds the `increment` routine in the example, which is repeated below. Note that this class was not used by the psect that holds the code associated with `main`. As this example program contains no instructions that *call* the `main` label (one instruction *jumps* to it) this routine can be linked anywhere in program memory and can continue to use the `code` psect (which is defined by the assembler to use the `CODE` class).

```
PSECT entryCode,class=ENTRY,delta=2
increment:
    movwf   loc
    incf    loc,w
    return
```

After building the example using the command shown in 9.2. Building the Example, you should see something similar to the following exert taken from the generated map file. Highlighted are the linker options the PIC Assembler driver automatically generates and how the psects used in the example were linked based on those options.

```
...
  -Mmain.map -E1 --acfsm=1493 -ACODE=00h-01FFhx4 -ASTRCODE=00h-07FFh \
  -AENTRY=00h-0FFh-01FFh,0200h-02FFh-03FFh,0400h-04FFh-05FFh,0600h-06FFh-07FFh \
...
TOTAL           Name                                 Link     Load   Length     Space
        CLASS   CODE
                resetVec                                0        0        5         0
                code                                 1F6      1F6        A         0
        CLASS   ENTRY
                entryCode                              5        5        3         0
```

You can see in the linker options at the top that the `CODE` class consists of four blocks of memory, each being an entire page in size (0x1FF). The `ENTRY` class consists of four blocks, where each block spans an entire page but with an entry range defined over the first 0xFF bytes of each page.

Down further, you can see that the `code` psect happened to be linked within the top half of a memory page (at address 0x1F6), but the `entryCode` psect was linked within one of the entry ranges of the `ENTRY` class (at address 0x5), as was requested.

The linker will find it more difficult to position psects into a class such as `ENTRY` due to the additional restrictions imposed by that class, but linking them with these restrictions is necessary for correct program operation. If routines do not have entry points that are called or reached via an instruction that modifies the PCL register, consider placing

them in a psect that is linked into the regular `CODE` class (such as the `code` psect), to utilize the potentially unused upper portions of program memory pages.

9.2 Building the Example

If the entire example source code for this chapter was saved in a plain text file, called `incPort.S`, for example, it could be built using the command below. Note that the file name uses an upper case `.S` extension so that the file is preprocessed before any other processing.

```
pic-as -mcpu=16F570 -Wl,-presetVec=0h -Wa,-a -Wl,-Map=incPort.map incPort.S
```

The psect containing the reset code has been linked to address 0. The command also includes the options to generate a map file (`incPort.map`). The option, `-Wa,-a`, generates an assembly list file (`incPort.lst`), so you can explore the generated code.

If you are building in the MPLAB X IDE, add additional linker options (such as those specified using the driver option `-Wl,`*`linkerOption`*) to the **Custom linker options** field in the **pic-as Linker** > **general** category in the Project Properties dialog. When adding options to this field, do not include the leading `-Wl,`, as this prefix is added in by the IDE. Specify additional options to the assembler (such as the `-Wa,`*`assemblerOption`*) in the **Additional options** field in the **pic-as Global Options** category (the leading `-Wa,` must be specified). See 3.7. Building the Example for screen captures showing where these additional options should be added.

10. Document Revision History

Revision A (May 2020)

- Initial release of this document.

Revision B (January 2021)

- Duplication of and updates to interrupt example code.
- Change of device and addition of configuration bits to Mid-range examples.
- Corrected operator used with banking examples.
- Added MPLAB X IDE text for building and screen captures for project property settings.
- Minor code and text corrections and improvements.

Revision C (June 2022)

- Added new chapter for Baseline devices, highlighting entry ranges in linker classes.
- Quoted configuration bit directives in code examples, which is now possible with the latest assembler release.
- Added information relating to the new linker option that performs automatic truncation of instruction operands.
- Corrected `reloc` value specified in the code example for PIC18 devices using vectored interrupts.
- Numerous but minor text improvements and clarifications.

The Microchip Website

Microchip provides online support via our website at www.microchip.com/. This website is used to make files and information easily available to customers. Some of the content available includes:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip design partner program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Product Change Notification Service

Microchip's product change notification service helps keep customers current on Microchip products. Subscribers will receive email notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, go to www.microchip.com/pcn and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Embedded Solutions Engineer (ESE)
- Technical Support

Customers should contact their distributor, representative or ESE for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in this document.

Technical support is available through the website at: www.microchip.com/support

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip products:

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner, within operating specifications, and under normal conditions.
- Microchip values and aggressively protects its intellectual property rights. Attempts to breach the code protection features of Microchip product is strictly prohibited and may violate the Digital Millennium Copyright Act.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is "unbreakable". Code protection is constantly evolving. Microchip is committed to continuously improving the code protection features of our products.

Legal Notice

This publication and the information herein may be used only with Microchip products, including to design, test, and integrate Microchip products with your application. Use of this information in any other manner violates these terms. Information regarding device applications is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. Contact your local Microchip sales office for additional support or, obtain additional support at www.microchip.com/en-us/support/design-help/client-support-services.

Trademarks

ISBN: 978-1-6683-0581-2

Quality Management System

For information regarding Microchip’s Quality Management Systems, please visit www.microchip.com/quality.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: www.microchip.com/support Web Address: www.microchip.com **Atlanta** Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455 **Austin, TX** Tel: 512-257-3370 **Boston** Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088 **Chicago** Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075 **Dallas** Addison, TX Tel: 972-818-7423 Fax: 972-818-2924 **Detroit** Novi, MI Tel: 248-848-4000 **Houston, TX** Tel: 281-894-5983 **Indianapolis** Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Australia - Sydney** Tel: 61-2-9868-6733 **China - Beijing** Tel: 86-10-8569-7000 **China - Chengdu** Tel: 86-28-8665-5511 **China - Chongqing** Tel: 86-23-8980-9588 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 **China - Hong Kong SAR** Tel: 852-2943-5100 **China - Nanjing** Tel: 86-25-8473-2460 **China - Qingdao** Tel: 86-532-8502-7355 **China - Shanghai** Tel: 86-21-3326-8000 **China - Shenyang** Tel: 86-24-2334-2829 **China - Shenzhen** Tel: 86-755-8864-2200 **China - Suzhou** Tel: 86-186-6233-1526 **China - Wuhan** Tel: 86-27-5980-5300 **China - Xian** Tel: 86-29-8833-7252 **China - Xiamen** Tel: 86-592-2388138 **China - Zhuhai** Tel: 86-756-3210040	**India - Bangalore** Tel: 91-80-3090-4444 **India - New Delhi** Tel: 91-11-4160-8631 **India - Pune** Tel: 91-20-4121-0141 **Japan - Osaka** Tel: 81-6-6152-7160 **Japan - Tokyo** Tel: 81-3-6880- 3770 **Korea - Daegu** Tel: 82-53-744-4301 **Korea - Seoul** Tel: 82-2-554-7200 **Malaysia - Kuala Lumpur** Tel: 60-3-7651-7906 **Malaysia - Penang** Tel: 60-4-227-8870 **Philippines - Manila** Tel: 63-2-634-9065 **Singapore** Tel: 65-6334-8870 **Taiwan - Hsin Chu** Tel: 886-3-577-8366 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 **Thailand - Bangkok** Tel: 66-2-694-1351 **Vietnam - Ho Chi Minh** Tel: 84-28-5448-2100	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4485-5910 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-72400 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-72884388 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820