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

Notice to Customers

All documentation becomes dated and this manual is no exception. Microchip tools and documentation are constantly evolving to meet customer needs, so some actual dialogs and/or tool descriptions can differ from those in this document. Please refer to our web site (https://www.microchip.com) to obtain the latest documentation available.

Documents are identified with a “DS” number. This number is located on the bottom of each page, in front of the page number. The numbering convention for the DS number is “DSXXXXXA,” where “XXXXX” is the document number and “A” is the revision level of the document.

For the most up-to-date information on development tools, see the MPLAB® IDE online help. Select the Help menu, and then Topics to open a list of available online help files.

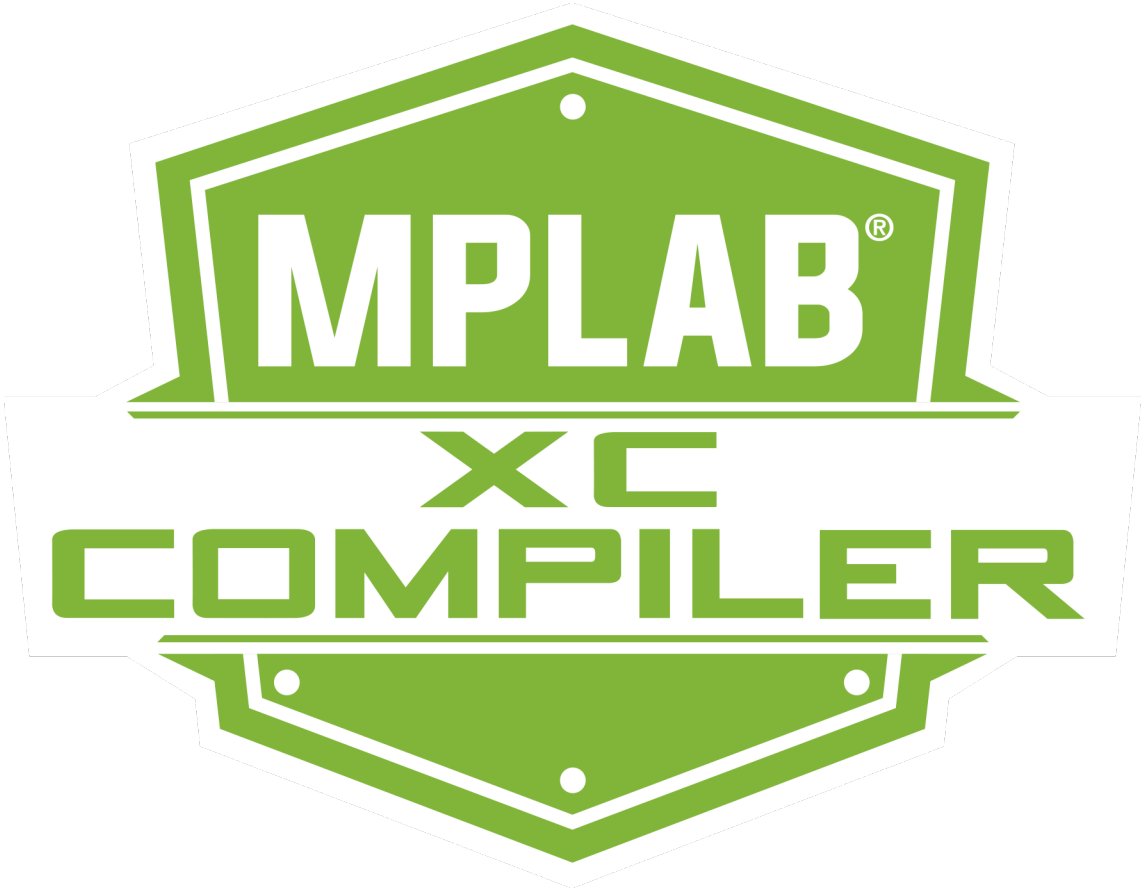

Table of Contents

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.

1.3 Document Revision History

Revision A (May 2020)

- Initial release of this document.

2. Introduction

This guide shows and describes example assembly programs that can be built with the MPLAB® XC8 PIC Assembler (PIC Assembler) for a variety of 8-bit Microchip PIC device families.

The examples shown bring together various concepts, assembler directives, and operators, which you can read about in more detail in the *MPLAB® XC8 PIC Assembler User's Guide* The programs themselves show how device or assembler features can be utilized but do not represent meaningful programs.

If you are migrating programs from the Microchip MPASM™ Assembler to 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 for a PIC18F47K42 device, which 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 and are mostly relevant to all devices.

A Basic PIC18 Example

```
PROCESSOR 18F47K42

#include <xc.inc>

 // configuration word 1
CONFIG FEXTOSC=XT           // crystal oscillator
CONFIG RSTOSC=EXTOSC        // EXTOSC operating per FEXTOSC bits
CONFIG CLKOUTEN=OFF         // CLKOUT function is disabled
CONFIG PR1WAY=ON            // PRLOCK bit can be cleared and set only once
CONFIG CSWEN=ON             // Writing to NOSC and NDIV is allowed
CONFIG FCMEN=ON             // Fail-Safe Clock Monitor enabled
// configuration word 2
CONFIG MCLRE=EXTMCLR        // If LVP=0, MCLR pin is MCLR; If LVP=1, RE3 pin function 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 the ZCDSEN bit of ZCDCON
CONFIG PPS1WAY=ON           // PPSLOCK cleared/set only once; 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 and Indexed Addressing Mode disabled
// configuration word 3
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; keyed access not required
CONFIG WDTCCS=SC            // Software Control
// configuration word 4
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 (300000-30000Bh) 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               // Low voltage programming enabled, MCLR pin, MCLRE ignored
// configuration word 5
CONFIG CP=OFF               // PFM and Data EEPROM code protection disabled

;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:
    clrf       max,c            ;starting point
    movff      max,ANSELA       ;write 0 to select digital input for port
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
```

as shown in 3. A Basic Example For PIC18 Devices, or they can be at the end of a line containing an instruction or directive, as shown bolded in this example:

```
    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 bolded 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 the example shown in 3. A Basic Example For PIC18 Devices, 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.

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, which is relevant only for C programs, but the setting-value pairs used are relevant for the `CONFIG` assembler directive.

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 `.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.

As shown in 3. A Basic Example For PIC18 Devices by the line:

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

your assembly source files will typically always include the `<xc.inc>` file, which is provided by the PIC Assembler. This file allows your source code to access special functions registers and other device features. Do not include device-specific include files 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 3. A Basic Example For PIC18 Devices and are discussed here.

Processor Directive

The `PROCESSOR` directive 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 there is a mismatch between the device specified in the `PROCESSOR` directive and in the option, an error will be triggered. The line of code:

```
PROCESSOR 18F47K42
```

shows the PIC18F47K42 device being specified.

Note that you can have sections of your program conditional on one or more devices 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 selected device. 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*.

In the examples shown in 3. A Basic Example For PIC18 Devices, the predefined `code` psect has been used to hold most of the program's code. You can see it being used in the lines:

```
PSECT code
main:
  clrf     max,c
  ...
```

thereby placing the `main` label and the instructions that follow into this psect. The `udata_acs` psect has been used to place `max` and `tmp` in the Access Bank memory, as shown in the lines:

```
PSECT udata_acs
max:
  DS    1
tmp:
  DS    1
```

Configuration data also needs to be in a psect; however, the `CONFIG` directive automatically places its data into an appropriate psect for you, so you should not precede it with a `PSECT` directive.

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.

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 3. A Basic Example For PIC18 Devices, 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 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.

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. As the `resetVec` psect will need to be explicitly linked to an absolute address, 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. 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 the instructions must be word aligned. Set it to 1 (the default value if no `reloc` flag is specified) for psects holding instructions for other devices.

The option that can be used to link the `resetVec` psect at the Reset vector location is given at the end of this section.

3.7 Building on the Command Line

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 3. A Basic Example For PIC18 Devices was saved in a plain text file, called `test.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 test.S
```

This command will produce (among other files) `test.hex` and `test.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.

Although the above command will build 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.

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

Open `test.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 lists, 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.

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 thru 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=test.map -Wl,-presetVec=0h test.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
```

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
```

4. A Basic Example For Mid-range Devices

The following example of a complete assembly program performs the same task as the code in 3. A Basic Example For PIC18 Devices, that is, it records the highest value present on PORTA, but this time for a PIC16F1937 device.

Many aspects of this code have already been covered in the PIC18 example, but other parts have been implemented differently and are discussed in the sections that follow.

A Basic PIC16 Example

```
PROCESSOR 16F1937

#include <xc.inc>

;configuration word 1
CONFIG FOSC=XT               // XT Oscillator
CONFIG WDTE=OFF,PWRTE=OFF    // WDT & PWRT disabled
CONFIG MCLRE=ON              // MCLR/VPP pin function is MCLR
CONFIG CP=OFF,CPD=OFF        // Program & data memory unprotected
CONFIG BOREN=ON              // Brown-out Reset enabled
CONFIG CLKOUTEN=OFF          // CLKOUT function is disabled
CONFIG IESO=ON               // Internal/External Switchover mode enabled
CONFIG FCMEN=ON              // Fail-Safe Clock Monitor enabled
;configuration word 2
CONFIG WRT=OFF               // Write protection off
CONFIG VCAPEN=OFF            // All VCAP pin functionality disabled
CONFIG PLLEN=ON              // 4x PLL enabled
CONFIG STVREN=ON             // Stack Over/underflow will cause a Reset
CONFIG BORV=LO               // Brown-out Reset Voltage low trip point
CONFIG LVP=ON                // Low-voltage programming enabled

skipnc  MACRO
    btfsc     STATUS,0
    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:
    PAGESEL    loop               ;ensure subsequent jumps are correct
    BANKSEL    max                ;starting point
    clrf       BANKMASK(max)
    BANKSEL    ANSELA             ;write 0 to select digital input for port
    clrf       BANKMASK(ANSELA)
loop:
    BANKSEL    PORTA              ;read and store port value
    movf       BANKMASK(PORTA),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 example program.

Assembler macros perform a similar function to the preprocessor's `#define` directive, but when the macro represents a large number of instructions or text spread over multiple lines, assembler macro definitions might be easier to read. In this case, the macro is defined to represent just one instruction, and is as shown here.

```
skipnc  MACRO
    btfsc   STATUS,0
    ENDM
```

The macro is used just the once in this example, shown here:

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

An assembler macro can have arguments, 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

As was done in 3. A Basic Example For PIC18 Devices, this example also places the entry-point code associated with Reset in a user-defined psect. 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 be 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 on those devices should be set to 1 (the default value if no `delta` flag is used).

Any program memory psect holding 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 PIC18 code shown in 3. A Basic Example For PIC18 Devices avoided data memory banking complications by using the `movff` instruction, which could access the entire data memory on the device used 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.

The Mid-range and Baseline PIC devices discussed in this example, 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.

If all the objects in your program are located in the same bank, then only a single bank selection sequence might be required. Similarly, address masking can omitted for objects in bank 0. However, it is good practice to always use address masking and be particularly careful of bank selection, especially if you modify the placement of objects.

The main part of the example program begins as follows.

```
PSECT code
main:
  BANKSEL max
  clrf    BANKMASK(max)
  BANKSEL ANSELA
  clrf    BANKMASK(ANSELA)
loop:
  BANKSEL PORTA
  movf    BANKMASK(PORTA),w
  BANKSEL tmp
  movwf   BANKMASK(tmp)
  ;is PORTA larger than input?
  subwf   max^(tmp&0ff80h),w
  ...
```

The `BANKSEL(max)` directive has been used to select the bank of the object `max` before it is accessed by the 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, 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 information is not removed, the linker may issue a fixup overflow error.

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 WREG 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 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. 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 example code 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 known 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.

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 not use the `BANKMASK()` macro.

4.4 Building the Example

If the entire source code shown in 4. A Basic Example For Mid-range Devices was saved in a plain text file, called `test.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=16f1937 -Wl,-presetVec=0h -Wa,-a -Wl,-Map=test.map test.S
```

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

5. Multiple Source Files, Paging and Linear Memory Example

In this PIC16F1937 example, the code reads 100 values from PORTA and stores these into the elements of an array accessed using linear addressing code. Linear memory access is only implement on Enhanced Mid-range devices, but other aspects of this code are applicable for all devices.

The 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 paging.

PIC16 Example - file_1.S

```
PROCESSOR 16F1937

#include <xc.inc>

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

;object in common memory
GLOBAL count                          ;make this globally accessible
PSECT udata_shr
count:
    DS        1

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

GLOBAL storeLevel                     ;link in with global symbol defined elsewhere

PSECT code
main:
    BANKSEL   ANSELA
    clrf      BANKMASK(ANSELA)
    BANKSEL   TRISC
    movlw     0ffh
    movwf     BANKMASK(TRISC)
    clrf      count
loop:
    ;a call to a routine in same psect
    call      readPort             ;value return in WREG
    ;a call to a routine in different module
    PAGESEL   storeLevel
    call      storeLevel           ;expects argument in WREG
    PAGESEL   $
    ;wait for a few cycles
    movlw     3
delay:
    decfsz    WREG,f
    goto      delay
    ;increment the array index, count, and stop iterating
    ;when the final element is reached
    movlw     100
    incf      count,f
    xorwf     count,w
    btfss     ZERO
    goto      loop
    goto      main

    END    resetVec
```

PIC16 Example - file_2.S

```
PROCESSOR 16F1937

#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 100 bytes of linear memory, at banked address 0x120
DLABS  1,0x120,100,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
    movwf     FSR1L              ;array, levels, storing
    movlw     high(levels)       ;the result in FSR1
    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 symbol in every other file that needs to access it and tell 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 5. Multiple Source Files, Paging and Linear Memory Example, `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
```

As `count` should 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 this symbol may be accessed from other modules.

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

The same mechanism is used for symbols used by routines that need to be accessible from more than one module. In the example code , the code in `file_2.S` contains a `GLOBAL storeLevel` directive to allow the routine to be called from outside that module, and `file_1.S` also contains a `GLOBAL storelevel` directive to link in with this global symbol. 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. Psects are grouped and concatenated based on how and where they are defined. This grouping may affect how you write code that jumps to labels or calls other routines.

Notice in `file_1.S` shown in 5. Multiple Source Files, Paging and Linear Memory Example that there were two separate blocks of assembly code placed into the `code` psect. 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`.

Once you have built the example program, the map file will show two `code` psects linked at different addresses. As shown in the map file extract below, 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`. (They are also shown in the listing by class, but there you cannot see the source file in which they were defined.) Note that there is just the one `code` psect indicated for `file_1.o`, and that the `code` psect from `file_2.o` was actually linked before the `code` psect from the other module.

```
                 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.

Before making a call or jump on Baseline and Mid-range devices, the PCLATH register must contain the value 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` in the example code, repeated here:

```
loop:
    call       readPort
    PAGESEL    storeLevel
    call       storeLevel
    PAGESEL    $
    ...
    goto      delay
```

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

As shown above, the call to `readPort` did not use a `PAGESEL` directive. 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 16F1937 device used in this example, the top of the map file is as follows:

```
Linker command line:

-W-3 --edf=/Applications/microchip/XC8/v2.10/pic/dat/en_msgs.txt -cs \
  -h+file_1.sym --cmf=file_1.cmf -z -Q16F1937 \
  -o/tmp/xc1oFI5Ul -Mfile_1.map -E1 -ver=XC8 --acfsm=1493 \
  -ASTACK=02000h-0218Bh -pstack=STACK -ACODE=00h-07FFhx4 \
  ...
```

The CODE class is defined by the linker option: -ACODE=00h-07FFhx4. This option indicates that the memory associated with the CODE class consists of 4 consecutive pages, the first starting at address 0, and each being 0x800 words long. These ranges correspond to the page addresses on the 16F1937 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, but 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.

There are two pseudo instructions that you can use to ensure that page selection always takes place. They 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:
    call       readPort
    fcall      storeLevel
    ...
    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.

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 (rather than have them linked anywhere in a linker class) to control where page selection instructions are needed, 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 as one contiguous and linear block on Enhanced Mid-range devices. 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. The definition of objects larger than a single bank is different to that for ordinary objects accessed only through banked memory.

The example code defines an object that is larger than a bank using the `DLABS` directive, shown here.

```
;define 100 bytes of linear memory, at banked address 0x120
DLABS  1,0x120,100,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 is 0x20A0, for example. The following argument is the number of bytes that is to be reserved, and the final argument is a symbol. The symbol 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 inside any psect. 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.

For this example, the compiler will reserve storage for the `levels` object in banks 2 and 3. Using banked instructions to access it, however, is problematic, because it may not be clear which bank to select beforehand. The Enhanced Mid-range devices, however, allow you to use the FSR register to indirectly read and write data memory, and accessing `levels` in this way is shown in the example and is repeated here.

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

5.4 Building the Example

If the two source code examples shown in 5. Multiple Source Files, Paging and Linear Memory Example (together with additional code to configure the device and peripherals) were saved in plain text files called `file_1.S` and `file_2.S`, they could be built using the command below. Note that both files have been specified in the one command and the assembler only needs to be executed just the once.

```
pic-as -mcpu=16f1937 -Wl,-presetVec=0h -Wa,-a -Wl,-Map=test.map file_1.S file_2.S
```

The psect containing the reset code has been linked to address 0. The command also includes the options to generate a map (`test.map`) and assembly list file (`test.lst`), so you can explore the generated 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=16f1937 -c file_1.S
pic-as -mcpu=16f1937 -c file_2.S
pic-as -mcpu=16f1937 -Wl,-presetVec=0h -Wa,-a -Wl,-Map=test.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 options that are passed to the linker and those that create the list and map files are only needed in the final build step. The `-mcpu` option, which indicates the target device, must be used with every command.

6. Compiled Stack Example

The example code in this section 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 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 PIC18 device is shown below. Configuration bit settings and the setup of the peripherals have been omitted for clarity.

A Compiled Stack Example

```
#include <xc.inc>

;place the compiled stack in Access bank memory
;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

;write needs one 2-byte parameter
FNSIZE write,0,2
GLOBAL ?pa_write
;write the low byte of the argument to LATA; the high byte to LATB
write:
    movff      ?pa_write+0,LATA
    movff      ?pa_write+1,LATB
    return

FNROOT main                    ;this is the root of a call graph
FNSIZE main,4,0                ;two 2-byte 'autos'
FNCALL main,add                ;main calls add
FNCALL main,write              ;and write
GLOBAL ?au_main
PSECT code
main:
    clrf       ?au_main+0,c             ;intermediate result
    clrf       ?au_main+1,c
```

```
    movlw      2                        ;increment amount
    movwf      ?au_main+3,c
loop:
    movff      ?au_main+0,?pa_add+0     ;load 1st parameter for add
    movff      ?au_main+1,?pa_add+1
    movff      ?au_main+2,?pa_add+2     ;load 2nd parameter for add
    movff      ?au_main+3,?pa_add+3
    call       add
    movff      ?pa_add+0,?au_main+0     ;store add's return value
    movff      ?pa_add+1,?au_main+1
    movff      ?au_main+0,?pa_write+0   ;load the parameter for write
    movff      ?au_main+1,?pa_write+1
    call       write
    infsnz     ?au_main+2,f,c           ;increase the increment value
    incf       ?au_main+3,f,c
    goto       loop

    END        resetVec
```

The function of the above assembly code is similar to that of the below C code, which is less complex and will be more familiar to C programmers.

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

void write(int val) {
    LATA = val;
    LATB = val >> 8;
}

int result, incr;

void main(void) {
    result = 0;
    incr = 2;
    while(1) {
        result = add(result, incr);
        write(result);
        incr++;
    }
}
```

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 (Access bank data memory) will be used to hold all the objects on the stack. The psect you choose 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 a data bank. 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 compiler stack on these devices.

The `?au_` symbol specified as the second directive argument will be prefixed to the name of each routine that uses the stack to create the base symbol for their auto-style objects. The `?pa_` prefix will be used to form the base symbol for each routine's parameter objects. These symbols are illustrated below.

The `add` routine begins with the following code:

```
PSECT code
FNSIZE add,0,4
```

```
GLOBAL ?pa_add
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 typical to have it 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 be used to create one, 4-byte-wide object if required.

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

```
FNROOT main
FNSIZE main,4,0
FNCALL main,add
FNCALL main,write
GLOBAL ?au_main
PSECT code
main:
    clrf    ?au_main+0,c
```

The main routine requires four bytes of auto objects and so the FNSIZE directive has again been used to indicate this. You can see the first byte of main's auto area accessed by the crlf instruction, using the symbol ?au_main.

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. 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 write. 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 example, 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 and ?au_main+1. You may, of course, define macros to make these expressions more readable, for example:

```
#define  add_b  ?pa_add+2
```

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
     write 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 `write`. 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 `write` are identical. This implies that the memory they use is shared, which is possible because `add` and `write` 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 memory usage.

The `-mcallgraph` option can be used to customize what call graph information in 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 corruption of data due to 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. For this program, it might look something like the following.

```
                                  Symbol Table

?au_main        udata_acs   0000  ?pa_add         udata_acs   0004
?pa_write       udata_acs   0004  __Hcode         code        0000
...
```

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 just happened to be linked to address 0, so the APB used by `main` for auto-style objects begins at address 0 (the same as its offset) and is shown to be in the `udata_acs` psect (as we specified). The APB used by `add` for its parameters begins at address 4, as does the block used by `write`.

6.3 Building the Example

If the entire source code shown in 6. Compiled Stack Example was saved in a plain text file, called `test.S` (together with additional code to configure the device and peripherals), 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=18F4520 -Wl,-presetVec=0h -Wa,-a -Wl,-Map=test.map -mcallgraph=full test.S
```

No special options are needed to build code that uses a compiled stack, but the above command uses the `-mcallgraph` option to request that a full callgraph be printed. The callgraph is shown in the map file, requested by the option `-Wl,-Map=test.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 option to generate an assembly list file (`test.lst`), so you can explore the generated code.

7. Interrupts and Bits Example

The following PIC18 device example code increments the value on PORTA every time a timer interrupt fires, and the source code for it is discussed in the sections that follow. In order to focus on certain aspects of the assembler, the configuration bits and timer setup, for example, are not present in this example.

An Example of Interrupts and Bits

```
PROCESSOR 18F44K22

#include <xc.inc>

;bits in bank 0 data memory
GLOBAL needCopy
PSECT myFlags,space=1,class=BANK0,bit
needCopy:     DS      1

;context save object
PSECT udata_bank0
shadow:       DS      1

PSECT udata_bank1
count:        DS      1

;Reset vector code
PSECT resetVec,class=CODE,reloc=2
resetVec:
        goto      main

;Interrupt vector code
PSECT intCodeLo,class=CODE,reloc=2
intVecLo:
        movff      BSR,shadow              ;save used registers
        btfsc      TMR0IE                  ;did the timer generate this interrupt?
        btfss      TMR0IF
        goto       exitInt                 ;no - ignore
        bcf        TMR0IF                  ;yes - clear the interrupt flag
        movlw      0FEh                    ;reload the timer
        movwf      TMR0
        ;set a flag to say that the port should be incremented
        BANKSEL    (needCopy/8)
        bsf        BANKMASK(needCopy/8),needCopy&7
exitInt:
        movff      shadow,BSR              ;restore context
        retfie

;main program code
PSECT code
main:
        ;other code to initialize the system goes here
        BANKSEL    (needCopy/8)
        ;enable interrupts
        bsf        PEIE
        bsf        GIE
        BANKSEL    (needCopy/8)
loop:
        ;do we need to increment yet?
        btfss      BANKMASK(needCopy/8),needCopy&7
        goto       loop                    ;no - keep checking
        ;yes - increment the value on the port
        movff      count,PORTA
        BANKSEL    count
        incf       BANKMASK(count),f
        BANKSEL    needCopy/8              ;reset the copy flag
        bcf        BANKMASK(needCopy/8),needCopy&7
        goto       loop

        END        resetVec
```

7.1 Interrupt Code

This example uses an interrupt handler. The entry point to interrupt code, like that associated with the Reset vector, must be linked to a specific address determined by the target device. To allow this to take place, the interrupt routine, or at least the entry point to the interrupt routine, should be placed in a unique psect that can be linked to the required address, as seen in the example code repeated here, which uses the `intCodeLo` psect to hold the entire routine.

```
PSECT intCodeLo,class=CODE,reloc=2
intVecLo:
        movff   BSR,shadow      ;save used registers
        btfsc   TMR0IE          ;did the timer generate this interrupt?
        btfss   TMR0IF
        goto    exitInt         ;no - ignore
        bcf     TMR0IF          ;yes - clear the interrupt flag
        movlw   0FEh            ;reload the timer
        movwf   TMR0
        ;set a flag to say that the port should be incremented
        BANKSEL (needCopy/8)
        bsf     BANKMASK(needCopy/8),needCopy&7
exitInt:
        movff   shadow,BSR      ;restore context
        retfie
```

Interrupt routines must save and restore the state of any registers that they (and any routine they call) use and that are also used in main-line code. With some devices, commonly-used registers are automatically saved into shadow registers when the interrupt occurs, but this is not always the case. Check your device datasheet to see how interrupts are processed on your device. In this example, the interrupt routine itself needs to save context, as it will be linked to the PIC18 low priority interrupt vector and the fast register stack was not used.

In this example, the only register used by both interrupt and main-line code is the BSR register, which holds the selected data bank. That register is modified by the `BANKSEL` directives. This register, therefor needs to be saved on entry to the interrupt routine and restored on exit. The WREG and STATUS registers usually need to be saved; however, in this example, they are not used by both routines.

Registers that must be saved should be copied to objects that you must define. In this example, an object called `shadow` was created for that purpose. Note that it was written and read using a `movff` instruction, which does not affect the STATUS or BSR registers. Be careful that your context save code itself does not clobber registers that have not yet been saved, and that your context restore code does not clobber register that have already been restored.

A `retfie` instruction was used to return from the interrupt.

7.2 Defining And Using Bits

In the interrupt routine shown in this example, a flag set by the interrupt code was used by the main-line code to determine when the port needed to be incremented. As this flag only needs to hold a true or false state, it was made a bit object to save on storage space.

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 `needCopy` is defined by the following lines:

```
PSECT myFlags,space=1,class=BANK0,bit
needCopy:     DS      1
```

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` directive, which allocates one unit of storage, is reserving a single bit, not a single byte. The `needCopy` object, then, is an object that can only hold a single bit.

Any label defined within a bit psect represents a bit address, not a byte address, and this affects how these labels should be used in instructions. The PIC instructions that work with bits require a byte address operand followed by a bit offset 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 PIC bit instruction, such as `bcf` or `btfss`, then you will need to divide the bit address by 8 to obtain the byte address used

in the instruction, and you will need to take the modulus of 7 to obtain the bit offset. For example, to set needCopy, the example code used:

```
BANKSEL (needCopy/8)
bsf     BANKMASK(needCopy/8),needCopy&7
```

Note that the BANKSEL directive also requires a byte address argument, so the bit address of needCopy was divided by 8 for this instruction as well.

You can, if desired, create a preprocessor macro to make this bit object more readable, for example

```
#define  NEEDCOPY  BANKMASK(needCopy/8),needCopy%7
```

which could then be used as follows:

```
bsf NEEDCOPY
```

You will notice that bits with SFRs are defined and accessed in just this way, for example in the instruction:

```
btfsc   TMR0IE
```

the TMR0IE macro expands to the byte address and bit offset. All the SFR bit symbols are actually preprocessor macros, which are defined in a similar way to the NEEDCOPY macro above. These SFR bits are available once you include <xc.inc> into your source file.

Note also that the addresses 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 for 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 this example, the myFlags psect was shown:

```
                Name                 Link     Load   Length Selector   Space Scale
interrupts_pic18.o
                intCodeLo              18       18       1C        C       0
                resetVec                0        0        2        0       0
                myFlags               308       61        1       60       1     8
```

A bit psect can be linked anywhere in the data memory with no restriction. In this example, it was associated with the BANK0 linker class, which means that it was placed somewhere in the bank 0 data memory. If you need to fast access bit objects, try to place them in the Access bank on PIC18 devices, or in common memory on other devices.

7.3 Building the Example

If the entire source code shown in 7. Interrupts and Bits Example (together with additional code to configure the device and peripherals) was saved in a plain text file, called test.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=18F44K22 -Wl,-presetVec=0h -Wl,-pintCodeLo=018h -Wa,-a -Wl,-Map=test.map test.S
```

The psect containing the reset code has been linked to address 0 and in addition, the psect holding the low priority interrupt code was linked to the device's vector location 0x18. The command also includes the options to generate a map (test.map) and assembly list file (test.lst), so you can explore the generated code.

The Microchip Website

Microchip provides online support via our website at http://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 http://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: http://www.microchip.com/support

Product Identification System

To order or obtain information, e.g., on pricing or delivery, refer to the factory or the listed sales office.

PART NO.	[X](1)	-	X	/XX	XXX
Device	Tape and Reel Option		Temperature Range	Package	Pattern

Device:	PIC16F18313, PIC16LF18313, PIC16F18323, PIC16LF18323	
Tape and Reel Option:	Blank	= Standard packaging (tube or tray)
	T	= Tape and Reel(1)
Temperature Range:	I	= -40°C to +85°C (Industrial)
	E	= -40°C to +125°C (Extended)
Package:(2)	JQ	= UQFN
	P	= PDIP
	ST	= TSSOP
	SL	= SOIC-14
	SN	= SOIC-8
	RF	= UDFN
Pattern:	QTP, SQTP, Code or Special Requirements (blank otherwise)	

Examples:

- PIC16LF18313- I/P Industrial temperature, PDIP package
- PIC16F18313- E/SS Extended temperature, SSOP package

Note:

1. Tape and Reel identifier only appears in the catalog part number description. This identifier is used for ordering purposes and is not printed on the device package. Check with your Microchip Sales Office for package availability with the Tape and Reel option.
2. Small form-factor packaging options may be available. Please check http://www.microchip.com/packaging for small-form factor package availability, or contact your local Sales Office.

Microchip Devices Code Protection Feature

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

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Legal Notice

Information contained in this publication regarding device applications and the like 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. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

The Microchip name and logo, the Microchip logo, Adaptec, AnyRate, AVR, AVR logo, AVR Freaks, BesTime, BitCloud, chipKIT, chipKIT logo, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, HELDO, IGLOO, JukeBlox, KeeLoq, Kleer, LANCheck, LinkMD, maXStylus, maXTouch, MediaLB, megaAVR, Microsemi, Microsemi logo, MOST, MOST logo, MPLAB, OptoLyzer, PackeTime, PIC, picoPower, PICSTART, PIC32 logo, PolarFire, Prochip Designer, QTouch, SAM-BA, SenGenuity, SpyNIC, SST, SST Logo, SuperFlash, Symmetricom, SyncServer, Tachyon, TempTrackr, TimeSource, tinyAVR, UNI/O, Vectron, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

APT, ClockWorks, The Embedded Control Solutions Company, EtherSynch, FlashTec, Hyper Speed Control, HyperLight Load, IntelliMOS, Libero, motorBench, mTouch, Powermite 3, Precision Edge, ProASIC, ProASIC Plus, ProASIC Plus logo, Quiet-Wire, SmartFusion, SyncWorld, Temux, TimeCesium, TimeHub, TimePictra, TimeProvider, Vite, WinPath, and ZL are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, BlueSky, BodyCom, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, EtherGREEN, In-Circuit Serial Programming, ICSP, INICnet, Inter-Chip Connectivity, JitterBlocker, KleerNet, KleerNet logo, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, SAM-ICE, Serial Quad I/O, SMART-I.S., SQI, SuperSwitcher, SuperSwitcher II, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

The Adaptec logo, Frequency on Demand, Silicon Storage Technology, and Symmcom are registered trademarks of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

ISBN: 978-1-5224-6126-5

AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Quality Management System

For information regarding Microchip's Quality Management Systems, please visit http://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: http://www.microchip.com/support Web Address: http://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