# MPLAB® XC8 C Compiler User's Guide for AVR® MCU

**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.



*Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.*

**QUALITY MANAGEMENT SYSTEM**
**CERTIFIED BY DNV**
**═ ISO/TS 16949 ═**

**Trademarks**



ISBN: 978-1-5224-2969-2

# Table of Contents



# Preface

## 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 (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.**

## INTRODUCTION

This chapter contains general information that will be useful to know before using the MPLAB® XC8 C Compiler User's Guide. Items discussed in this chapter include:

- Document Layout
- Conventions Used in this Guide
- Recommended Reading
- The Microchip Web Site
- Development Systems Customer Change Notification Service
- Customer Support
- Document Revision History

## DOCUMENT LAYOUT

The MPLAB XC8 C Compiler User's Guide is organized as follows:

- Chapter 1. Compiler Overview
- Chapter 2. XC8 Command-line Driver
- Chapter 3. C Language Features
- Chapter 4. Utilities
- Appendix A. Library Functions
- Appendix B. Implementation-Defined Behavior
- Glossary
- Index

## CONVENTIONS USED IN THIS GUIDE

This manual uses the following documentation conventions:

**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) { ... }` |

## RECOMMENDED READING

This user's guide describes how to use MPLAB XC8 C Compiler. Other useful documents are listed below. The following Microchip documents are available and recommended as supplemental reference resources.

**Readme for MPLAB XC8 C Compiler**

For the latest information on using MPLAB XC8 C Compiler, read *MPLAB® XC8 C Compiler Release Notes* (an HTML file) in the Docs subdirectory of the compiler's installation directory. The release notes contain update information and known issues that cannot be included in this user's guide.

**Readme Files**

For the latest information on using other tools, read the tool-specific Readme files in the Readmes subdirectory of the MPLAB IDE installation directory. The Readme files contain update information and known issues that cannot be included in this user's guide.

## THE MICROCHIP WEB SITE

Microchip provides online support via our web site at www.microchip.com. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **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 consultant 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

## DEVELOPMENT SYSTEMS CUSTOMER CHANGE NOTIFICATION SERVICE

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

To register, access the Microchip web site at www.microchip.com, click on **Customer Change Notification** and follow the registration instructions.

The Development Systems product group categories are:

- **Compilers** – The latest information on Microchip C compilers, assemblers, linkers and other language tools. These include all MPLAB C compilers; all MPLAB assemblers (including MPASM™ assembler); all MPLAB linkers (including MPLINK™ object linker); and all MPLAB librarians (including MPLIB™ object librarian).
- **Emulators** – The latest information on Microchip in-circuit emulators.This includes the MPLAB REAL ICE™ and MPLAB ICE 2000 in-circuit emulators.
- **In-Circuit Debuggers** – The latest information on the Microchip in-circuit debuggers. This includes MPLAB ICD 3 in-circuit debuggers and PICkit™ 3 debug express.
- **MPLAB® IDE** – The latest information on Microchip MPLAB IDE, the Windows® Integrated Development Environment for development systems tools. This list is focused on the MPLAB IDE, MPLAB IDE Project Manager, MPLAB Editor and MPLAB SIM simulator, as well as general editing and debugging features.
- **Programmers** – The latest information on Microchip programmers. These include production programmers such as MPLAB REAL ICE in-circuit emulator, MPLAB ICD 3 in-circuit debugger and MPLAB PM3 device programmers. Also included are non production development programmers such as PICSTART® Plus and PICkit 2 and 3.

## CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or field application engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the web site at:
http://www.microchip.com/support

## DOCUMENT REVISION HISTORY

### Revision A (April 2018)

Initial release of this document, adapted from the MPLAB XC8 C Compiler User's Guide, DS50002053.

**NOTES:**

# Chapter 1. Compiler Overview

## 1.1 INTRODUCTION

This chapter is an overview of the MPLAB® XC8 C Compiler for AVR devices, including these topics.

- Compiler Description and Documentation
- Device Description

## 1.2 COMPILER DESCRIPTION AND DOCUMENTATION

The MPLAB XC8 C Compiler is a free-standing, optimizing ISO C99 compiler. It supports all 8-bit PIC® and AVR microcontrollers; however, this document describes how to use the compiler when targeting Microchip AVR devices. See the MPLAB® XC8 C Compiler User's Guide for PIC® MCU for information on using this compiler when targeting Microchip PIC devices.

> **Note:** Features described as being part of MPLAB XC8 in this document assume that you are using a Microchip AVR device. These features may differ if you choose to instead compile for a Microchip PIC device.

The compiler is available for several popular operating systems, including Professional editions of Microsoft Windows 7 (32/64 bit), Windows 8 (64 bit), and Windows 10 (64 bit); Ubuntu 16.04 (32/64 bit); Fedora 23 (64 bit) or Mac OS X 10.12 (64 bit).

The compiler is available in two operating modes: Free or PRO. The PRO operating mode is a licensed mode and requires an activation key to enable it. Free mode is available for unlicensed customers. The basic compiler operation, supported devices and available memory are identical across all modes. The modes only differ in the level of optimization employed by the compiler.

### 1.2.1 Conventions

Throughout this manual, the term "compiler" is used. It can refer to all, or a subset of the collection of applications that comprise the MPLAB XC8 C Compiler. When it is not important to identify which application performed an action, it will be attributed to "the compiler."

In a similar manner, "compiler" is often used to refer to the command-line driver; although specifically, the driver for the MPLAB XC8 C Compiler package is named `xc8-cc` (as discussed in Section 2.7 "Option Descriptions"). Accordingly, "compiler options" commonly refers to command-line driver options.

In a similar fashion, "compilation" refers to all or a selection of steps involved in generating an executable binary image from source code.

## 1.3 DEVICE DESCRIPTION

This compiler guide describes the MPLAB XC8 compiler's support for all 8-bit Microchip AVR devices, including tinyAVR, and AVR XMEGA devices.

The compiler takes advantage of the target device's instruction set, addressing modes, memory, and registers whenever possible. A summary of the device families is shown in Table 1-1. This includes the offset of the special function registers, the address at which general-purpose data memory starts, and the offset at which program memory is mapping into the data space (where relevant).

**TABLE 1-1: SUMMARY OF SUPPORTED DEVICE FAMILIES**

| Family | ArchID | SFR Offset | Mapped Flash Address |
|---|---|---|---|
| avr1 | 1 | 0x20 | n/a |
| avr2 | 2 | 0x20 | n/a |
| avr25 | 25 | 0x20 | n/a |
| avr3 | 3 | 0x20 | n/a |
| avr31 | 31 | 0x20 | n/a |
| avr35 | 35 | 0x20 | n/a |
| avr4 | 4 | 0x20 | n/a |
| avr5 | 5 | 0x20 | n/a |
| avr51 | 51 | 0x20 | n/a |
| avr6 | 6 | 0x20 | n/a |
| avrtiny | 100 | 0x0 | 0x4000 |
| avrxmega2 | 102 | 0x0 | n/a |
| avrxmega3 | 103 | 0x0 | 0x8000 |
| avrxmega4 | 104 | 0x0 | n/a |
| avrxmega5 | 105 | 0x0 | n/a |
| avrxmega6 | 106 | 0x0 | n/a |
| avrxmega7 | 107 | 0x0 | n/a |

# Chapter 2. XC8 Command-line Driver

## 2.1 INTRODUCTION

The name of the MPLAB XC8 command-line driver is `xc8-cc`. This driver can be invoked to perform all aspects of compilation, including C code generation, assembly, and link steps. It is the recommended way to use the compiler, as it hides the complexity of all the internal applications and provides a consistent interface for all compilation steps.

If you are building a legacy project or would prefer to use the old command-line driver you may instead run the `avr-gcc` driver application and use appropriate command-line options for that driver. Those options may differ to those described in this guide.

This chapter describes the steps that the driver takes during compilation, the files that the driver can accept and produce, as well as the command-line options that control the compiler's operation.

The following topics are examined in this chapter of the MPLAB XC8 C Compiler User's Guide:

- Invoking the Compiler
- The Compilation Sequence
- Runtime Files
- Compiler Output
- Compiler Messages
- Option Descriptions

## 2.2 INVOKING THE COMPILER

This section explains how to invoke `xc8-cc` on the command line, as well as the files that it can read.

### 2.2.1 Driver Command-line Format

The `xc8-cc` driver has the following basic command format:

`xc8-cc [options] files [libraries]`

Throughout this manual, it is assumed that the compiler applications are in the console's search path (see Section 2.2.2 "Driver Environment Variables") or that the full path is specified when executing an application.

It is customary to declare *options* (identified by a leading dash "-" or double dash "--") before the files' names. However, this is not mandatory.

The formats of the options are supplied in Section 2.7 "Option Descriptions" along with corresponding descriptions of the options.

The *files* can be an assortment of C and assembler source files, and precompiled intermediate files. While the order in which these files are listed is not important, it can affect the allocation of code or data, and can affect the names of some of the output files.

*Libraries* is a list of user-defined library files that will be searched by the compiler, in addition to the standard C libraries. The order of these files will determine the order in which they are searched. It is customary to insert the *Libraries* list after the list of source file names; however, this is not mandatory.

If you are building code using a make system, see Section 2.3.3 "Multi-Step Compilation".

#### 2.2.1.1 LONG COMMAND LINES

The `xc8-cc` driver can be passed a command-line file containing driver options and arguments to circumvent any operating-system-imposed limitation on command line length.

A command file is specified by the `@` symbol, which should be immediately followed (i.e., no intermediate space character) by the name of the file containing the arguments.

Inside the file, each argument must be separated by one or more spaces and can extend over several lines. The file can contain blank lines, which will be ignored.

The following is the content of a command file, `xyz.xc8` for example, that was constructed in a text editor and that contains the options and the file names required to compile a project.

```
-mcpu=atmega3290p -Wl,-Map=proj.map -Wa,-a=proj.lst
-O2 main.c isr.c
```

After this file is saved, the compiler can be invoked with the following command:

```
xc8-cc @xyz.xc8
```

Command files can be used as a simple alternative to a make file and utility, as well as conveniently stored compiler options and source file names.

### 2.2.2 Driver Environment Variables

No environment variables are defined or required by the compiler for it to execute.

Adjusting the PATH environment variable allows you to run the compiler driver without having to specify the full compiler path.

This variable can be automatically updated when installing the compiler by selecting the **Add xc8 to the path environment variable** checkbox in the appropriate dialog.

Note that the directories specified by the PATH variable are only used to locate the compiler driver. Once the driver is running, it will manage access to the internal compiler applications, such as the assembler and linker, etc.

The MPLAB X IDE allows the compiler to be selected via the Project properties dialog without the need for the PATH variable.

### 2.2.3 Input File Types

The xc8-cc driver accepts a number of input file types, which are distinguished by the file's extension, unless the -x language option (Section 2.7.2.6 "x: Specify Source Language") is specified. The recognized input file extensions are listed in Table 2-1.

**TABLE 2-1: xc8-cc INPUT FILE TYPES**

| Extension | File format |
|---|---|
| .c | C source file |
| .i | Preprocessed C source file |
| .s | Assembler source file |
| .S or .sx | Assembly source file to be preprocessed |
| .o | Relocatable object code file |
| other | A file to be passed to the linker |

There are no compiler restrictions imposed on the names of source files, but be aware of case, name-length, and other restrictions that are imposed by your operating system.

Note that the invocation of the C preprocessor will be automatic when an assembly source file is listed on the command line using a .S extension (capital letter "S"). This is not dependent on the operating system on which the compiler is running, since the compiler checks the extension present on the command line, not the extension of the source file name.

Avoid using the same base name for assembly and C source files, even if they are located in different directories, and avoid having source files with the same basename as the MPLAB X IDE project name.

## 2.3 THE COMPILATION SEQUENCE

When you compile a project, many internal applications are called to do the work. This section looks at when these internal applications are executed and how this relates to the build process of multiple source files. This section should be of particular interest if you are using a make system to build projects.

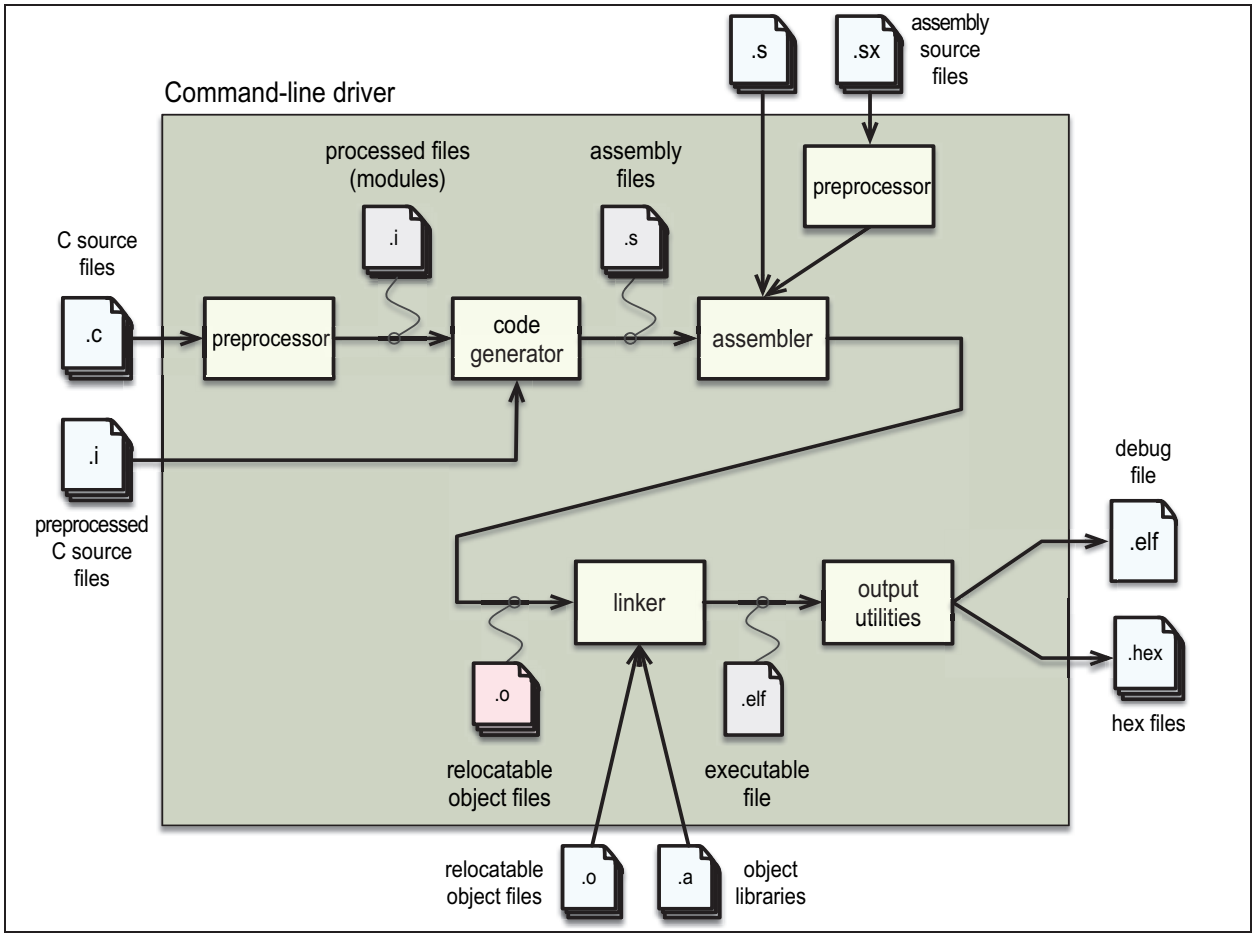
### 2.3.1 The Compiler Applications

The main internal compiler applications and files are illustrated in Figure 2-1.

You can consider the large underlying box to represent the whole compiler, which is controlled by the command line driver, `xc8-cc`. You can be satisfied just knowing that C source files (shown on the far left) are passed to the compiler and the resulting output files (shown here as a HEX and ELF debug file—by default called `a.out`—on the far right) are produced; however, internally there are many applications and temporary files being produced. An understanding of the internal operation of the compiler, while not necessary, does assist with using the tool.

The driver will call the required compiler applications when required. These applications are located in the compiler's bin directories and are shown in the diagram as the smaller boxes inside the driver. The temporary files produced by each application can also be seen in this diagram and are marked at the point in the compilation sequence where they are generated. The intermediate files for C source are shaded in red. Some of these temporary files remain after compilation has concluded. There are also driver options to request that the compilation sequence halt after execution of a particular application so that the output of that application remains in a file and can be examined.

**FIGURE 2-1: COMPILER APPLICATIONS AND FILES**



### 2.3.2 Single-Step Compilation

Compilation of one or more source files can be performed in just one step using the `xc8-cc` driver.

The following command will build both C source files and the assembly source file, passing the files to the appropriate internal applications, then link the generated output to form the final output.

```
xc8-cc -mcpu=at86rf401 main.c io.c mdef.s
```

The driver will compile all source files, regardless of whether they have changed since the last build. Development environments (such as MPLAB® X IDE) and make utilities must be employed to achieve incremental builds (as described in Section 2.3.3 “Multi-Step Compilation”).

Unless otherwise specified, an ELF file (this is by default called `a.out`) is produced as the final output. The intermediate files and most other temporary files are deleted after each build, unless you use the `-save-temps` option (see Section 2.7.5.3 “save-temps”). Note that if you are using the MPLAB X IDE, some generated files can be placed in a different directory to where your project source files are located.

### 2.3.3 Multi-Step Compilation

A multi-step compilation method can be employed to achieve an incremental build of your project. Make utilities take note of which source files have changed since the last build and only rebuild these files to speed up compilation. From within MPLAB X IDE, you can select an incremental build (Build Project icon), or fully rebuild a project (Clean and Build Project icon).

Make utilities typically call the compiler multiple times: once for each source file to generate an intermediate file, and once to perform the second stage compilation.

The option `-c` (see Section 2.7.2.1 “c: Compile to Intermediate File”) is used to create an intermediate file. The option stops compilation after the assembler has executed, and the resulting assembly output file will have a `.o` extension.

The intermediate files are then specified to the driver during the second stage of compilation, when they will be passed to the linker.

The first two of the following command lines build an intermediate file for each C source file, then these intermediate files are passed to the driver again to complete the compilation in the last command.

```
xc8-cc -mcpu=ata6289 -c main.c
xc8-cc -mcpu=ata6289 -c io.c
xc8-cc -mcpu=ata6289  main.o io.o
```

As with any compiler, all the files that constitute the project must be present when performing the second stage of compilation, thus you cannot, for example, generate an HEX file from only part of the project.

You might also wish to generate intermediate files to construct your own library files. See Section 4.2 “Librarian” for more information on library creation.

### 2.3.4 Compilation of Assembly Source

Assembly files are compiled in a similar way to C source files. The compiler driver knows that these files should be passed to the assembler first.

If you wish to use the C preprocessor to parser an assembly source file for preprocessor directives, ensure the source file uses a `.S` or `.sx` extension, for example `init.sx`.

## 2.4 RUNTIME FILES

In addition to the C and assembly source files specified on the command line, there are also pre-compiled library and object files that the compiler can link into your project. These files are discussed in the following sections.

### 2.4.1 Library Files

The names of the C standard library files appropriate for the selected target device are determined by the driver and automatically passed to the linker. You do not need to manually include the compiler-supplied library files into your project.

The standard libraries, such as `libc.a` are found in the `avr/avr/lib` directory. Emulation routines for operations not natively supported in hardware are part of `libgcc.a`, found in `avr/lib/gcc/avr/`. The `libm.a` math library is also automatically linked in, as is `lib`*`devicename`*`.a` (e.g. `libatxmega128b1.a`) that contains device-specific routines for working with watch dog timers, power management, eeprom access, etc., (see Section 3.11 "Libraries" for more information).

### 2.4.2 Startup and Initialization

Pre-built object files, which contain the runtime startup code, are provided with the compiler. Section 3.10.2 "Runtime Startup Code" details the specific actions taken by this code and how it interacts with programs you write.

The runtime startup code is executed before `main`. However, if you require any special initialization to be performed immediately after Reset, you should write a powerup initialization routine (described later in Section 3.10.3 "The Powerup Routine").

## 2.5 COMPILER OUTPUT

There are many files created by the compiler during the compilation. A large number of these are temporary files. Many are deleted after compilation is complete, but some remain and are used for programming the device, or for debugging purposes.

> **Note:** Throughout this manual, the term *project name* will refer to the name of the project created in the IDE.

### 2.5.1 Output Files

The default behavior of `xc8-cc` is to produce an ELF file called `a.out` file.

The name of the output file can be changed using the `-o` option (see Section 2.7.2.3 "o: Specify Output File"). Compiler options can alter the compilation process and generate different output file types. The common output file types are shown inTable 2-2.

**TABLE 2-2: COMMON OUTPUT FILES**

| Extension | Format and method of creation |
|---|---|
| `.hex` | Intel HEX file, produced by execution of `avr-objcopy` |
| `a.out`, `.elf` | ELF/Dwarf, generated by default |
| `.o` | Relocatable object (intermediate) file, produced by `-c` |
| `.s` | Assembly file, produced by `-S` |
| `.i` | Preprocessed C file, produced by `-E` |

A HEX file must be created by calling the `avr-objcopy` application. This application is automatically called when you are using MPLAB X IDE, once to generate the main output, and again to generate the HEX file for EEPROM data.

The ELF file is used by debuggers to obtain debugging information about the project.

### 2.5.2 Diagnostic Files

Two valuable files produced by the compiler are the assembly list file generated by the assembler and the map file generated by the linker. They are output by options, shown in Table 2-3.

**TABLE 2-3: DIAGNOSTIC FILES**

| File format | Type and option used |
|---|---|
| `file.lst` | Assembly list file, produced by `-Wa,-a=file.lst` |
| `file.map` | Map file, produced by `-Wl,-Map=file.map` |

See Section 4.4 “Objdump” for more information on generating these files.

## 2.6 COMPILER MESSAGES

All compiler applications, including the command-line driver, `xc8-cc`, use textual messages to report feedback during the compilation process.

### 2.6.1 Message Type

There are three types of messages (described below). The behavior of the compiler when encountering a message of each type is also listed.

| | |
|---|---|
| **Warning Messages** | Indicates source code or some other situation that can be compiled, but is unusual and may lead to runtime failures of the code. The code or situation that triggered the warning should be investigated; however, compilation of the current module will continue, as will compilation of any remaining modules. |
| **Error Messages** | Indicates source code that is illegal or that compilation of this code cannot take place. Compilation will be attempted for the remaining source code in the current module, but no additional modules will be compiled and the compilation process will then conclude. |
| **Fatal Error Messages** | Indicates a situation in which the compilation cannot proceed and requires that the compilation process to stop immediately. |

### 2.6.2 Changing Message Behavior

The attributes of individual messages can be modified during compilation using compiler options.

#### 2.6.2.1 DISABLING MESSAGES

By default, the compiler can issue a variety of warnings. All warning messages can be disabled by using `-w` option.

You can explicitly turn off warnings by using the `-Wno-`*`message`* option, where *`message`* relates to the warning type, for example `-Wno-return-type`. When a warning is produced by the compiler, it prints in square brackets the associated warning option that controls this warning. For example, if the compiler issues the warning:

```
avr.c:13:1: warning: 'keep' attribute directive ignored [-Wattributes]
```

you can disable this warning using the option `-Wno-attributes`.

> **Note:** Disabling warning messages in no way fixes the condition that triggered the message. Always use extreme caution when exercising these options.

You can enable a more complete set of warning messages about questionable constructions by using `-Wall`. The `-Wextra` option turns on additional messages. Alternatively, you can enable individual messages using the `-W`*`message`* option, for example `-Wunused-function` (see Section 2.7.4 "Options for Controlling Warnings and Errors").

#### 2.6.2.2 CHANGING MESSAGE TYPES

It is also possible to change the type of some messages. Warnings can be turned into errors by using the `-Werror` option. Errors can be turned into fatal errors by using the `-Wfatal-errors` option.

## 2.7 OPTION DESCRIPTIONS

Most aspects of the compilation can be controlled using the command-line driver, `xc8-cc`. The driver will configure and execute all required applications, such as the code generator, assembler and linker.

It is recommended, especially for options that control optimizations or other code generation aspects of the compiler, that you limit your use to only those options described in this user's guide, even if the options are described in the generic GCC documentation. Not all GCC options have been implemented or may not have been tested. In addition, some options use alternate forms to be compatible with other Microchip MPLAB XC compilers, for example the `-mcpu` option should be used in preference to the `-mmcu` GCC option, making this aspect of compilation for AVR devices compatible with other Microchip compilers.

All options are identified by single or double leading dash character, e.g. `-c` or `--version`. The options are case sensitive.

See the `--help` option, Section 2.7.2.8 "Help", for a brief description of accepted options.

If you are compiling from within the MPLAB X IDE, it will by default, issue explicit options to the compiler (unless changed in the Project Properties dialog) and these options can be different to those that are the default on the command line.

If you are compiling the same project from the command line and from the MPLAB X IDE, always check that you explicitly specify each option.

The following categories of options are described.

- Options Specific to AVR Devices
- Options for Controlling the Kind of Output
- Options for Controlling the C Dialect
- Options for Controlling Warnings and Errors
- Options for Debugging
- Options for Controlling Optimization
- Options for Controlling the Preprocessor
- Options for Assembling
- Options for Linking
- Options for Directory Search
- Options for Code Generation Conventions

### 2.7.1 Options Specific to AVR Devices

The options shown in Table 2-4 are useful when compiling for 8-bit Microchip AVR devices with the MPLAB XC8 compiler and are discussed in the sections that follow.

**TABLE 2-4: AVR DEVICE-SPECIFIC OPTIONS**

| Option | Controls |
|---|---|
| `-m[no-]accumulate-args` | How arguments are passed between functions |
| `-m[no-]call-prologues` | How functions save and restore registers |
| `-mcpu=`*device* | The target device or architecture for which to compile |
| `-mno-interrupts` | How the stack pointer is changed |
| `-fno-jump-tables` | Whether jump tables are used in `switch()` statements |
| `-mrelax` | Optimization of call/jump instructions |
| `-mshort-calls` | How function calls are encoded |
| `-mstrict-X` | The use of the X register |
| `-mtiny-stack` | The width of the stack pointer |

#### 2.7.1.1 ACCUMULATE-ARGS

The `-maccumulate-args` option prevents function arguments from being pushed onto and popped off the stack, instead producing code that adjusts the stack pointer once at the beginning of the calling function. This option has no effect when functions that do not use the stack for arguments are called, but for other functions, it can reduce code size if those functions are called several times.

#### 2.7.1.2 CALL-PROLOGUES

The `-mcall-prologues` option changes how functions save registers on entry and how those registers are restored on function exit.

If this option is not used or the `-mno-call-prologues` options is used, the registers that need to be preserved by each function will be saved and restored by code inside those functions. If the `-mcall-prologues` option is used, this preservation code is extracted as subroutines that are called at the appropriate points in the function.

Use of this option can reduce code size, but can increase the code's execution time.

#### 2.7.1.3 CPU

The `-mcpu` option should be used to specify the target device or at least a target architecture family. The available architecture families are shown in Table 2-5.

**TABLE 2-5: SELECTABLE ARCHITECTURE FAMILIES**

| Architecture | Architecture Features |
|---|---|
| `avr1` | Simple core, no data RAM, assembly support only |
| `avr2` | Classic core, up to 8kB program memory |
| `avr25` | avr2 with `MOVW` and `LPM Rx,Z[+]` instructions |
| `avr3` | Classic core with up to 64kB extended program memory |
| `avr31` | Classic core with 128kB of program memory |
| `avr35` | avr3 with `MOVW` and `LPM Rx,Z[+]` instructions |
| `avr4` | Enhanced core up to 8kB program memory |
| `avr5` | Enhanced core up to 64kB program memory |
| `avr51` | Enhanced core 128kB program memory |
| `avr6` | Enhanced core 256kB program memory |
| `avrxmega2` | XMEGA core, up to 64kB program memory, up to 64kB data address space |
| `avrxmega3` | xmega2 devices with program memory mapped in data address space |
| `avrxmega4` | XMEGA core, up to 128kB program memory, up to 64kB data address space |
| `avrxmega5` | XMEGA core, up to 128kB program memory, greater than 64kB data address space |
| `avrxmega6` | XMEGA core, greater than 128 kB program memory, up to 64kB data address space |
| `avrxmega7` | XMEGA core, greater than 128 kB program memory, greater than 64kB data address space |
| `avrtiny` | AVR Tiny core, 16 registers |

For example:

```
xc8-cc -mcpu=atmega161 main.c
```

#### 2.7.1.4 NO-INTERRUPTS

The `-mno-interrupts` option controls whether interrupts should be disabled when the stack pointer is changed.

For most devices, the state of the status register, SREG, is saved in a temporary register, and interrupts are disabled before the stack pointer is adjusted. The status register is then restored after the stack pointer has been changed.

If a program does not use interrupts, there is no need for the stack adjustments to be protected in this way. Use of this option omits the code that disables and potentially re-enables interrupts around the code the adjusts the stack pointer, thus reducing code size and execution time.

Since the AVR XMEGA devices and devices with an 8-bit stack pointer can change the value held by the stack pointer atomically, this option is not required and has no effect when compiling for one of these devices.

Specifying this option will define the preprocessor macro `__NO_INTERRUPTS__` to the value 1.

#### 2.7.1.5 NO-JUMP-TABLES

See Section 3.6.3 “Switch Statements” for details regarding the `-mno-jump-tables` option.

#### 2.7.1.6 RELAX

The `-mrelax` option controls the optimization of the long form of call and jump instructions, which are always output by the code generator into shorter and/or faster relative calls and jumps at link-time. These changes can only take place if the relative forms of the instructions can be determined to be in range of their destination (for more information see Section 3.4.6.3 "Function Pointers").

#### 2.7.1.7 SHORT-CALLS

The `-mshort-call` option controls how calls are encoded.

When building for devices which have more then 8kB of program memory, the compiler will automatically use the longer form of the jump and call instructions when program execution is leaving the current function. Doing so allows program execution to reach the entire memory space, but the program will be larger and take longer to execute. The `-mshort-calls` option will force calls to use PC-relative instructions such as the `rjmp` and `rcall` instructions, which have a limited destination range. This option has no effect on indirect jumps or calls made via function pointers.

Use this option with caution, as your code might fail if functions fall out of range of the shorter instructions. See the `-mrelax` option (Section 2.7.1.6 "relax") to allow function pointers to be encoded as 16-bits wide, even on large memory device. This option has no effect for the avr2 and avr4 architectures, which have less than 8kB of program memory and which always use the shorter form of the call/jump instructions.

#### 2.7.1.8 STRICT-X

The `-mstrict-X` option ensures that the X register (r26-r27) is only used in indirect, post-increment or pre-decrement addressing. This restricts the register's usage, which could be beneficial in terms of code size.

#### 2.7.1.9 TINY-STACK

The `-mtiny-stack` option controls the width of the stack pointer.

On some devices that have a small amount of data RAM, the stack pointer is only 8-bits wide. For other devices, it is 16-bits wide and occasionally each byte might need to be accessed separately to change where the stack pointer points.

If your devices uses a 16-bit stack pointer and the stack is located in the lower half of memory and is smaller than 256 bytes in size, this option will force the stack pointer to use only a single byte, thus reducing the amount of code necessary to adjust the stack pointer.

The option is automatically applied if the device RAM totals 256 bytes or less.

### 2.7.2 Options for Controlling the Kind of Output

The options shown in Table 2-6 control the kind of output produced by the compiler and are discussed in the sections that follow.

**TABLE 2-6: KIND-OF-OUTPUT CONTROL OPTIONS**

| Option | Produces |
|---|---|
| `-c` | An intermediate file |
| `-E` | A preprocessed file |
| `-o` *file* | An output file with the specified name |
| `-S` | An assembly file |
| `-v` | Verbose compilation |
| `-x` *language* | Output after preprocessing all source files |
| `-###` | Command lines but with no execution of the compiler applications |
| `--help` | Help information only |
| `--version` | Compiler version information |

#### 2.7.2.1 C: COMPILE TO INTERMEDIATE FILE

The `-c` option is used to halt compilation after executing the assembler, leaving a relocatable object intermediate file with a `.o` extension as the output.

It is frequently used when building using a make utility. See Section 2.3.3 “Multi-Step Compilation” for more information on generating and using intermediate files.

#### 2.7.2.2 E: PREPROCESS ONLY

The `-E` option is used to generate preprocessed C source files (also called modules or translation units).

When the `-E` option is used, the compilation sequence will terminate after the preprocessing stage. The preprocessed output is printed to `stdout`, but you can use the `-o` option to redirect this to a file.

You might check the preprocessed source files to ensure that preprocessor macros have expanded to what you think they should. The option can also be used to create C source files that do not require any separate header files. This is useful when sending files to a colleague or to obtain technical support without sending all the header files, which can reside in many directories.

#### 2.7.2.3 O: SPECIFY OUTPUT FILE

The -o option specifies the name and directory of the output file.

The option `-o main.elf`, for example, will place the compiler output in a file called `main.elf`, rather than the default file called `a.out`. The name of an existing directory can be specified with the file name, for example `-o build/main.elf`, so that the file will appear in that directory.

#### 2.7.2.4 S: COMPILE TO ASSEMBLY

The `-S` option stops compilation after generating an assembly output file.

The command:

```
xc8-cc -mcpu=atxmega32d4 -S test.c io.c
```

will produce two assembly file called `test.s` and `io.s`, which contain the assembly code generated from their corresponding source files.

This option might be useful for checking assembly code output by the compiler without the distraction of line number and opcode information that will be present in an assembly list file produced by the `-Wa,-a` option (see Section 2.7.9 "Mapped Assembler Options").

#### 2.7.2.5 V: VERBOSE COMPILATION

The `-v` option specifies verbose compilation.

When this option is used, the name and path of the executed compiler applications (described in Section 2.3 "The Compilation Sequence"), will be displayed, followed by the command-line arguments to this application, for example:

```
c:/program
files/atmel/studio/7.0/toolchain/avr8/avr8-gnu-toolchain/bin/../libexe
c/gcc/avr/5.4.0/cc1.exe -quiet -v -imultilib avr6 -iprefix c:\program
files\atmel\studio\7.0\toolchain\avr8\avr8-gnu-toolchain\bin\../lib/gc
c/avr/5.4.0/ avr2.c -mn-flash=4 -mno-skip-bug -quiet -dumpbase avr2.c
-mmcu=avr6 -auxbase avr2 -version -o avr2.s
```

#### 2.7.2.6 X: SPECIFY SOURCE LANGUAGE

The `-x` *`language`* option specifies the language for the following sources files.

The compiler usually uses the extension of an input file to determine the file's content. This option allows you to have the language of a file explicitly stated. The option remains in force until the next `-x` option, or the `-x none` option, which turns off the language specification entirely for subsequent files. The allowable languages are shown in Table 2-7.

**TABLE 2-7: SOURCE FILE LANGUAGE**

| Language | File language |
|---|---|
| `assembler` | Assembly source |
| `assembler-with-cpp` | Assembly with C preprocessor directives |
| `c` | C source |
| `cpp-output` | Preprocessed C source |
| `c-header` | C header file |
| `none` | Based entirely on the file's extension |

The `-x assembler-with-cpp` language option ensures assembly source files are preprocessed before they are assembled, thus allowing the use of preprocessor directives, such as `#include`, and C-style comments with assembly code. By default, assembly files not using a `.S` or `.sx` extension are not preprocessed.

You can create precompiled header files with this option, for example:

```
xc8-cc -mmcu=atxmega32d4 -x c-header init.h
```

will create the precompiled header called `init.h.gch`.

#### 2.7.2.7 ###

The `-###` option is similar to `-v`, but the commands are not executed. This option allows you to see the compiler's command lines without executing the compiler.

#### 2.7.2.8 HELP

The `--help` option displays information on the `xc8-cc` compiler options, then the driver will terminate.

#### 2.7.2.9 VERSION

The `--version` option prints compiler version information then exits.

### 2.7.3 Options for Controlling the C Dialect

The options shown in Table 2-8 define the type of C dialect used by the compiler and are discussed in the sections that follow.

**TABLE 2-8: C DIALECT CONTROL OPTIONS**

| Option | Controls |
|---|---|
| `-ansi` | Strict ANSI conformance |
| `-aux-info filename` | The generation of function prototypes |
| `-fno-asm` | Keyword recognition |
| `-fno-builtin`<br>`-fno-builtin-`*`function`* | Use of built-in functions |
| `-f[no-]signed-char`<br>`-f[no-]unsigned-char` | The signedness of a plain `char` type. |
| `-f[no-]signed-bitfields`<br>`-f[no-]unsigned-bitfields` | The signedness of a plain `int` bit-field. |
| `-mext=`*`extension`* | Which language extensions is in effect |
| `-std=`*`standard`* | The C language standard |

#### 2.7.3.1 ANSI

The `-ansi` option ensures C program strictly conform to the C90 standard.

When specified, this option turns off certain GCC language extensions when compiling C source, such as C++ style comments, keywords such as `asm` and `inline`. The macro `__STRICT_ANSI__` is defined when this option is in use. See also `-Wpedantic` for information on ensuring strict ISO compliance.

#### 2.7.3.2 AUX-INFO

The `-aux-info` option generates function prototypes from a C module.

The `-aux-info main.pro` option, for example, prints to `main.pro` prototyped declarations for all functions declared and/or defined in the module being compiled, including those in header files. Only one source file can be specified on the command line when using this option so that the output file is not overwritten. This option is silently ignored in any language other than C.

Besides declarations, the file indicates, in comments, the origin of each declaration (source file and line), whether the declaration was implicit, prototyped or unprototyped (`I`, `N` for new or `O` for old, respectively, in the first character after the line number and the colon), and whether it came from a declaration or a definition (`C` or `F`, respectively, in the following character). In the case of function definitions, a K&R-style list of arguments followed by their declarations is also provided, inside comments, after the declaration.

For example, compiling with the command:

```
xc8-cc -mmcu=atmega8a -aux-info test.pro test.c
```

might produce `test.pro` containing the following declarations, which can then be edited as necessary:

```
/* test.c:2:NC */ extern int add (int, int);
/* test.c:7:NF */ extern int rv (int a); /* (a) int a; */
/* test.c:20:NF */ extern int main (void); /* () */
```

#### 2.7.3.3 NO-ASM

The `-fno-asm` option restricts the recognition of certain keywords, freeing them to be used as identifiers.

When used, this option ensures that `asm`, `inline` and `typeof` are not recognized as keywords. You can, instead, use the keywords `__asm__`, `__inline__` and `__typeof__`.

The `-ansi` option implies `-fno-asm`.

#### 2.7.3.4 NO-BUILTIN

The `-fno-builtin` option will prevent the compiler from producing special code for built-in functions that do not begin with `__builtin_` as prefix.

Normally special code that avoids a function call is produced for many built-in functions. The resulting code is often both smaller and faster, but since the function calls no longer appear as such, you cannot set a breakpoint on those calls, nor can you change the behavior of the functions by linking with a different library.

The `-fno-builtin-`*`function`* form of this option allows you to prevent a built-in version of the named function from being used. In this case, *`function`* must not begin with `__builtin_`.

#### 2.7.3.5 SIGNED-CHAR/UNSIGNED-CHAR

The `-fsigned-char` and `-funsigned-char` options enforce the signedness of a plain `char` type.

By default, the plain `char` type is equivalent to `signed char`, unless the `-mext=cci` option has been used, in which case it is equivalent to `unsigned char`. These options specify the type that will be used by the compiler for plain `char` types. Using the `-funsigned-char` or the `-fno-signed-char` option forces a plain `char` to be unsigned, and the `-fsigned-char` or the `-fno-unsigned-char` option forces a plain `char` to be signed.

Consider explicitly stating the signedness of objects when they are defined, rather than relying on the type assigned to a plain `char` type by the compiler.

#### 2.7.3.6 SIGNED-BITFIELDS/UNSIGNED-BITFIELDS

The `-fsigned-bitfield` and `-funsigned-bitfield` options control the signedness of a plain `int` bit-field type.

By default, the plain `int` type, when used as the type of a bit-field, is equivalent to `signed int`. These options specify the type that will be used by the compiler for plain `int` bit-fields. Using the `-funsigned-bitfield` or the `-fno-signed-bitfield` option forces a plain `int` to be unsigned, and the `-fsigned-bitfield` or the `-fno-unsigned-bitfield` option forces a plain `int` to be signed.

Consider explicitly stating the signedness of bit-fields when they are defined, rather than relying on the type assigned to a plain `int` bit-field type.

#### 2.7.3.7 EXT

The `-mext=`*`extension`* option controls the language extensions allowed during compilation. The possible extensions arguments are shown in Table 2-9.

**TABLE 2-9: ACCEPTABLE C LANGUAGE EXTENSIONS**

| Extension | C Language Description |
|---|---|
| `xc8` | No extensions (default) |
| `cci` | A common C interface acceptable by all MPLAB XC compilers |

Enabling the `cci` extension requests the compiler to check all source code and compiler options for compliance with the Common C Interface (CCI). Code that complies with this interface can be more easily ported across all MPLAB XC compilers. Code or options that do not conform to the CCI will be flagged by compiler warnings.

#### 2.7.3.8 STD

The `--std=`*`standard`* option specifies the C standard with which the compiler and C source code should conform when compiling. The allowable standards are shown in Table 2-10.

**TABLE 2-10: ACCEPTABLE C LANGUAGE STANDARDS**

| Standard | Supports |
|---|---|
| `c89` or `c90` | All ISO C90 programs |
| `c99` | All ISO C99 programs |

### 2.7.4 Options for Controlling Warnings and Errors

Warnings are diagnostic messages that report constructions that are not inherently erroneous, but that are risky or suggest there may have been an error.

You can request many specific warnings with options beginning `-W`; for example, `-Wimplicit`, to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning `-Wno-` to turn off warnings; for example, `-Wno-implicit`. This guide lists only one of the two forms, whichever is not the default.

The options shown in Table 2-11 control the messages produced by the compiler and are discussed in the sections that follow.

**TABLE 2-11: WARNING AND ERROR OPTIONS IMPLIED BY ALL WARNINGS**

| Option | Controls |
|---|---|
| `-f[no-]syntax-only` | Checking code for syntax errors only |
| `-pedantic` | Warnings demanded by strict ANSI C; rejects all programs that use forbidden extensions |
| `-pedantic-errors` | Warnings implied by `-pedantic`, except that errors are produced rather than warnings |
| `-w` | Suppression of all warning messages |
| `-W[no-]all` | Enablement of all warnings |
| `-W[no-]address` | Warnings from suspicious use of memory addresses |
| `-W[no-]char-subscripts` | Warnings from array subscripts with type `char` |
| `-W[no-]comment` | Warnings from suspicious comments |
| `-W[no-]div-by-zero` | Warnings from compile-time integer division by zero. |
| `-Wformat` | Warnings from inappropriate `printf()` arguments |
| `-Wimplicit` | Warnings implied by both `-Wimplicit-int` and `-Wimplicit-function-declaration` |
| `-Wimplicit-function-declaration` | Warnings from use of undeclared function |
| `-Wimplicit-int` | Warnings from declarations not specifying a type |
| `-Wmain` | Warnings from unusual main definition |
| `-Wmissing-braces` | Warnings from missing braces |
| `-Wno-multichar` | Warnings from multi-character constant |
| `-Wparentheses` | Warnings from missing precedence |
| `-Wreturn-type` | Warnings from missing return type |
| `-Wsequence-point` | Warnings from sequence point violations |
| `-Wswitch` | Warnings from missing or extraneous case values |
| `-Wsystem-headers` | Warnings from code within system headers |
| `-Wtrigraphs` | Warnings from use of trigraphs |
| `-Wuninitialized` | Warnings from use of uninitialized variables |
| `-Wunknown-pragmas` | Warnings from use of unknown pragma |
| `-Wunused` | Warnings from unused objects and constructs |
| `-Wunused-function` | Warnings from unused static function |
| `-Wunused-label` | Warnings from unused labels |
| `-Wunused-parameter` | Warnings from unused parameter |
| `-Wunused-variable` | Warnings from unused variable |
| `-Wunused-value` | Warnings from unused value |

#### 2.7.4.1 SYNTAX ONLY

The `-fsyntax-only` option checks the C source code for syntax errors, then terminates the compilation.

#### 2.7.4.2 PEDANTIC

The `-pedantic` option ensures that programs do not use forbidden extensions and issues warnings when a program does not follow ISO C.

#### 2.7.4.3 PEDANTIC-ERRORS

The `-pedantic-errors` option works in the same way as the `-pedantic` option, only errors, instead of warnings, are issued when a program is not ISO compliant.

#### 2.7.4.4 W: DISABLE ALL WARNINGS

The `-w` option inhibits all warning messages.

#### 2.7.4.5 ALL

The `-Wall` option enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.

Note that some warning flags are not implied by `-Wall`. Some of them warn about constructions that users generally do not consider questionable, but which you might occasionally wish to check. Others warn about constructions that are necessary or hard to avoid in some cases and there is no simple way to modify the code to suppress the warning. Some of them are enabled by `-Wextra` but many of them must be enabled individually.

#### 2.7.4.6 ADDRESS

The `-Waddress` option will generate warnings about suspicious uses of memory addresses. These include using the address of a function in a conditional expression, such as `void func(void)`; if `(func)`, and comparisons against the memory address of a string literal, such as if `(x == "abc")`. Such uses typically indicate a programmer error: the address of a function always evaluates to true, so their use in a conditional usually indicates that the programmer forgot the parentheses in a function call; and comparisons against string literals result in unspecified behavior and are not portable in C, so they usually indicate that the programmer intended to use `strcmp`.

#### 2.7.4.7 CHAR-SUBSCRIPTS

The `-Wchar-subscripts` option will warn if an array subscript has type `char`.

#### 2.7.4.8 COMMENT

When using the `-Wcomment` option, the compiler will warn whenever a comment-start sequence `/*` appears in a `/*` comment, or whenever a backslash-newline appears in a comment started by `//`.

#### 2.7.4.9 DIV-BY-ZERO

The `-Wdiv-by-zero` option explicitly requests that the compiler warn about compile-time integer division by zero.To inhibit the warning messages, use `-Wno-div-by-zero`. No warnings will occur for floating-point division by zero, as it can be a legitimate way of obtaining infinities and NaNs.

#### 2.7.4.10 FORMAT

The `-Wformat` option checks calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified.

#### 2.7.4.11 IMPLICIT

The `-Wimplicit` option is equivalent to specifying both `-Wimplicit-int` and `-Wimplicit-function-declaration`.

#### 2.7.4.12 IMPLICIT-FUNCTION-DECLARATION

The `-Wimplicit-function-declaration` option will generate a warning when a function has been used but not declared.

#### 2.7.4.13 IMPLICIT-INT

The `-Wimplicit-int` option will generate a warning when a declaration does not specify a type.

#### 2.7.4.14 MAIN

The `-Wmain` option will generate a warning if the type of `main()` is suspicious. This function should be a function with external linkage, returning `int`, and typically taking no arguments. This option will allow you to define two or three arguments of appropriate types.

#### 2.7.4.15 MISSING-BRACES

The `-Wmissing-braces` option will warn if an aggregate or union initializer is not fully bracketed. In the following example, the initializer for `a` is not fully bracketed, while `b` is fully bracketed.

```
int a[2][2] = { 0, 1, 2, 3 };
int b[2][2] = { { 0, 1 }, { 2, 3 } };
```

#### 2.7.4.16 NO-MULTICHAR

The `-Wno-multichar` option will generate a warning if a multi-character `character` constant is used. Usually, such constants are typographical errors. Since they have implementation-defined values, they should not be used in portable code. The following example illustrates the use of a multi-character `character` constant:

```
char xx(void)
{
    return('xx');
}
```

#### 2.7.4.17 PARENTHESES

The `-Wparentheses` option will generate a warning if parentheses are omitted in certain situations, such as when there is an assignment in a context where a truth value is expected, or when operators are nested whose precedence people often find confusing. For example, the following will produce a warning, due to it not realizing that the `&&` operator has higher precedence.

```
if(a || b && c)
...
```

#### 2.7.4.18 RETURN-TYPE

The `-Wreturn-type` option will generate a warning whenever a function is defined with a return-type that is not specified and that defaults to `int`. The option will also trigger warnings about any `return` statement with no return value in a function whose return-type is not `void`.

#### 2.7.4.19 SEQUENCE-POINT

The `-Wsequence-point` option generates warnings when compiling code that may have undefined semantics because of violations of sequence point rules in the C standard.

The C standard defines the order in which expressions in a C program are evaluated in terms of sequence points, which represent a partial ordering between the execution of parts of the program: those executed before the sequence point and those executed after it. These occur after the evaluation of a full expression (one which is not part of a larger expression), after the evaluation of the first operand of a `&&`, `||`, `? :` or `,` (comma) operator, before a function is called (but after the evaluation of its arguments and the expression denoting the called function), and in certain other places. Other than as expressed by the sequence point rules, the order of evaluation of subexpressions of an expression is not specified. All these rules describe only a partial order rather than a total order. For example, if two functions are called within one expression with no sequence point between them, the order in which the functions are called is not specified. However, the standards committee has ruled that function calls do not overlap.

It is not specified when between sequence point modifications to the values of objects take effect. Programs whose behavior depends on this have undefined behavior. The C standard specifies that “Between the previous and next sequence point, an object shall have its stored value modified, at most once, by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.” If a program breaks these rules, the results on any particular implementation are entirely unpredictable.

Each of the following statements are examples that have undefined behavior:

```
a = a++;
a[n] = b[n++];
a[i++] = i;
```

Some more complicated cases are not diagnosed by this option, and it may give an occasional false positive result, but in general it has been found fairly effective at detecting this sort of problem in programs.

#### 2.7.4.20 SWITCH

The `-Wswitch` option generates warnings whenever a `switch` statement has an index of enumerated type and lacks a case value for one or more of the named codes of that enumeration. (The presence of a `default` label prevents this warning.) Any `case` label values outside the enumeration range also provoke warnings when this option is used.

#### 2.7.4.21 SYSTEM-HEADERS

The `-Wsystem-headers` option generates warnings for constructs found in system header files.

Warnings from system headers are normally suppressed on the assumption that they usually do not indicate real problems and would only make the compiler output harder to read. Using this command line option tells the compiler to emit warnings from system headers as if they occurred in user code. However, note that using `-Wall` in conjunction with this option does not warn about unknown pragmas in system headers. For that, `-Wunknown-pragmas` must also be used.

#### 2.7.4.22 TRIGRAPHS

The `-Wtrigraphs` option generates warnings if trigraphs are enabled and have been encountered in the program.

#### 2.7.4.23 UNINITIALIZED

The `-Wuninitialized` option generates warnings if an automatic variable is used without first being initialized. Such variables can contain unknown values.

These warnings are possible only when optimization is enabled, because they require data flow information that is computed only when optimizing.

These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared `volatile`, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers.

Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.

#### 2.7.4.24 UNKNOWN-PRAGMAS

The `-Wunknown-pragmas` option generates warnings when a `#pragma` directive is encountered which is not understood by the compiler. If this command line option is used, warnings will even be issued for unknown pragmas in system header files. This is not the case if the warnings were only enabled by the `-Wall` command line option.

#### 2.7.4.25 UNUSED

The `-Wunused` option generates warnings whenever the following is true:

- A variable is unused aside from its declaration
- A function is declared static but never defined
- A label is declared but not used
- A statement computes a result that is explicitly not used

In order to get a warning about an unused function parameter, both `-W` and `-Wunused` must be specified.

Casting an expression to void suppresses this warning for an expression. Similarly, the `unused` attribute suppresses this warning for unused variables, parameters and labels.

#### 2.7.4.26 UNUSED-FUNCTION

The `-Wunused-function` option generates warnings whenever a `static` function is declared but not defined or a non-inline `static` function is unused.

#### 2.7.4.27 UNUSED-LABEL

The `-Wunused-label` option generates warnings whenever a label is declared but not used. To suppress this warning, use the `unused` attribute.

#### 2.7.4.28 UNUSED-PARAMETER

The `-Wunused-parameter` option generates warnings whenever a function parameter is unused aside from its declaration. To suppress this warning, use the `unused` attribute.

#### 2.7.4.29 UNUSED-VARIABLE

The `-Wunusued-variable` option generates warnings whenever a local variable or non-constant `static` variable is unused aside from its declaration. To suppress this warning, use the `unused` attribute.

#### 2.7.4.30 UNUSED-VALUE

The `-Wunused-value` option generates warnings whenever a statement computes a result that is explicitly not used. To suppress this warning, cast the expression to `void`.

The following `-W` options are not implied by `-Wall`. Some of them warn about constructions that users generally do not consider questionable, but which you might occasionally wish to check for. Others warn about constructions that are necessary or hard to avoid in some cases, and there is no simple way to modify the code to suppress the warning.

**TABLE 2-12: WARNING OPTIONS NOT IMPLIED BY ALL WARNINGS**

| Option | Controls |
|---|---|
| `-Wextra` | The generation of additional warning messages |
| `-Waggregate-return` | Warnings from aggregate objects being returned |
| `-Wbad-function-cast` | Warnings from functions cast to a non-matching type |
| `-Wcast-qual` | Warnings from discarded pointer qualifiers |
| `-Wconversion` | Warnings from implicit conversions that can alter values |
| `-Werror` | Generation of errors instead of warnings for dubious constructs |
| `-Winline` | Warnings when functions cannot be in-lined |
| `-Wlarger-than=`*`len`* | Warnings when defining large objects |
| `-W[no-]long-long` | Warnings from use of `long long` |
| `-Wmissing-declarations` | Warnings when functions are not declared |
| `-Wmissing-format-attribute` | Warnings with missing format attributes |
| `-Wmissing-noreturn` | Warnings from potential `noreturn` attribute omissions |
| `-Wmissing-prototypes` | Warnings when functions are not declared with prototype |
| `-Wnested-externs` | Warnings from extern declarations |
| `-Wno-deprecated-declarations` | Whether warnings are produced for deprecated declarations |
| `-Wpointer-arith` | Warnings when taking size of unsized types |
| `-Wredundant-decls` | Warnings from redundant declarations |
| `-Wshadow` | Warnings when local objects shadow other objects |

**TABLE 2-12: WARNING OPTIONS NOT IMPLIED BY ALL WARNINGS**

| Option | Controls |
|---|---|
| -W[no-]sign-compare | Warnings from signed comparisons |
| -Wstrict-prototypes | Warnings from K&R function declarations |
| -Wtraditional | Warnings from traditional differences |
| -Wundef | Warnings from undefined identifiers |
| -Wunreachable-code | Warnings from unreachable code |
| -Wwrite-strings | Warnings when using non-const string pointers |

#### 2.7.4.31 EXTRA

The -Wextra option generates extra warnings in the following situations.

- A nonvolatile automatic variable might be changed by a call to longjmp. These warnings are possible only in optimizing compilation. The compiler sees only the calls to setjmp. It cannot know where longjmp will be called. In fact, a signal handler could call it at any point in the code. As a result, a warning may be generated even when there is in fact no problem, because longjmp cannot in fact be called at the place that would cause a problem.
- A function could exit both via return value; and return;. Completing the function body without passing any return statement is treated as return;.
- An expression-statement or the left-hand side of a comma expression contains no side effects. To suppress the warning, cast the unused expression to void. For example, an expression such as x[i,j] causes a warning, but x[(void)i,j] does not.
- An unsigned value is compared against zero with < or <=.
- A comparison like x<=y<=z appears. This is equivalent to (x<=y ? 1 : 0) <= z, which is a different interpretation from that of an ordinary mathematical notation.
- Storage-class specifiers like static are not the first things in a declaration. According to the C Standard, this usage is obsolescent.
- If -Wall or -Wunused is also specified, warn about unused arguments.
- A comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned (but won't warn if -Wno-sign-compare is also specified).
- An aggregate has a partly bracketed initializer. For example, the following code would evoke such a warning, because braces are missing around the initializer for x.h:

```
- struct s { int f, g; };
- struct t { struct s h; int i; };
- struct t x = { 1, 2, 3 };
```

- An aggregate has an initializer that does not initialize all members. For example, the following code would cause such a warning, because x.h would be implicitly initialized to zero:

```
- struct s { int f, g, h; };
- struct s x = { 3, 4 };
```

#### 2.7.4.32 AGGREGATE-RETURN

The -Waggregrate-return option generates warnings if any functions that return structures or unions are defined or called.

#### 2.7.4.33 BAD-FUNCTION-CAST

The `-Wbad-function-cast` option generates warnings whenever a function call is cast to a non-matching type. For example, warn if a call to a function defined as `int foof()` is cast to anything a pointer type.

#### 2.7.4.34 CAST-QUALIFIER

The `-Wcast-qual` option generates warnings whenever a pointer is cast, so as to remove a type qualifier from the target type. For example, warn if a `const char *` is cast to an ordinary `char *`.

#### 2.7.4.35 CONVERSION

The `-Wconversion` option generates warnings if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, as well as conversions changing the width or signedness of a fixed point argument, except when the same as the default promotion.

Also, warn if a negative integer constant expression is implicitly converted to an unsigned type. For example, warn about the assignment `x = -1` if `x` is unsigned. But do not warn about explicit casts like `(unsigned) -1`.

#### 2.7.4.36 ERROR

The `-Werror` option turns all warnings into errors.

#### 2.7.4.37 INLINE

The `-Winline` option generates warnings if a function can not be in-lined, and either it was declared as in-line, or else the `-finline-functions` option was given.

#### 2.7.4.38 LARGER-THAN

The `-Wlarger-than=`*`len`* option generates warnings whenever an object larger than *`len`* bytes is defined.

#### 2.7.4.39 LONG LONG

The `-Wlong-long` option generates warnings if `long long` types are used. This is the default. To inhibit the warning messages, use `-Wno-long-long`. Flags `-Wlong-long` and `-Wno-long-long` are taken into account only when `-pedantic` flag is used.

#### 2.7.4.40 MISSING-DECLARATIONS

The `-Wmissing-declarations` option generates a warning if a global function is defined without a previous declaration, even if that definition itself provides a prototype.

#### 2.7.4.41 MISSING-FORMAT-ATTRIBUTE

The `-Wmissing-format-attribute` option, when used with the `-Wformat` option, generates warnings about functions and function pointers that are candidates for the `format` attribute. Note that it is up to you to confirm that this attribute is valid for the indicated functions.

#### 2.7.4.42 MISSING-NORETURN

The `-Wmissing-noreturn` option generates warnings about functions that might be candidates for attribute `noreturn`. These are only possible candidates, not absolute ones. Care should be taken to manually verify functions. In fact, do not ever return before adding the `noreturn` attribute, otherwise subtle code generation bugs could be introduced.

#### 2.7.4.43 MISSING-PROTOTYPE

The `-Wmissing-prototypes` option generates warnings if a global function is defined without a previous prototype declaration, even if the definition itself provides a prototype. (This option can be used to detect global functions that are not declared in header files.)

#### 2.7.4.44 NESTED-EXTERNS

The `-Wnested-externs` option generates warnings if an `extern` declaration is encountered within a function.

#### 2.7.4.45 NO-DEPRECATED-DECLARATIONS

The `-Wno-deprecated-declarations` option prevents warnings about uses of functions, variables and types marked as deprecated by using the `deprecated` attribute.

#### 2.7.4.46 POINTER-ARITH

The `-Wpointer-arith` option generates warnings about code that depends on the size of a function type or of `void`. The compiler assigns these types a size of 1, for convenience in calculations with `void *` pointers and pointers to functions.

#### 2.7.4.47 REDUNDANT-DECLS

The `-Wredundant-decls` option generates warnings if anything is declared more than once in the same scope, even in cases where multiple declarations are valid and change nothing.

#### 2.7.4.48 SHADOW

The `-Wshadow` option generates warnings whenever a local variable shadows another local variable.

#### 2.7.4.49 SIGNED-COMPARE

The `-Wsigned-compare` option generates warnings when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. This warning is also enabled by `-W`. To get the other warnings of `-W` without this warning, use `-W -Wno-sign-compare`.

#### 2.7.4.50 STRICT-PROTOTYPES

The `-Wstrict-prototypes` option generates warnings if a function is declared or defined in the old K & R style, without specifying the argument types. (An old-style function definition is permitted without a warning if preceded by a declaration which specifies the argument types.)

#### 2.7.4.51 TRADITIONAL

The `-Wtraditional` option generates warnings about certain constructs that behave differently in traditional and ANSI C.

- Macro arguments occurring within string constants in the macro body. These would substitute the argument in traditional C, but are part of the constant in ANSI C.
- A function declared external in one block and then used after the end of the block.
- A switch statement has an operand of type long.
- A nonstatic function declaration follows a static one. This construct is not accepted by some traditional C compilers.

#### 2.7.4.52 UNDEF

The `-Wundef` option generates warnings if an undefined identifier is evaluated in an `#if` directive.

#### 2.7.4.53 UNREACHABLE-CODE

The `-Wunreachable-code` option generates warnings if the compiler detects code that will never be executed.

It is possible for this option to produce a warning even though there are circumstances under which part of the affected line can be executed, so care should be taken when removing apparently unreachable code. For instance, when a function is in-lined, a warning may mean that the line is unreachable in only one in-lined copy of the function.

#### 2.7.4.54 WRITE-STRING

The `-Wwrite-strings` option gives string constants the type `const char[length]` so that copying the address of one into a non-const `char *` pointer generates a warning. At compile time, these warnings help you find code that you can try to write into a string constant, but only if you have been very careful about using `const` in declarations and prototypes. Otherwise, it's just a nuisance, which is why `-Wall` does not request these warnings.

### 2.7.5 Options for Debugging

The options shown in Table 2-13 control the debugging output produced by the compiler and are discussed in the sections that follow.

**TABLE 2-13: DEBUGGING OPTIONS**

| Option | Controls |
|---|---|
| `-g` | The type of debugging information generated |
| `-Q` | Printing of diagnostics associated with each function as it is compiled, and statistics about each pass on conclusion. |
| `-save-temps`<br>`-save-temps=cwd`<br>`-save-temps=obj` | Whether intermediate files should be kept after compilation |

#### 2.7.5.1 G: PRODUCE DEBUGGING INFORMATION

The `-g` option instructs the compiler to produce additional information, which can be used by hardware tools to debug your program. By default, the compiler produces dwarf release 2 files.

The compiler supports the use of `-g` with `-O` making it possible to debug optimized code; however, the shortcuts taken by optimized code may occasionally produce surprising results. The following might be observed.

- Some declared variables may not exist at all
- Flow of control may briefly move unexpectedly
- Some statements may not be executed because they compute constant results or their values were already at hand
- Some statements may execute in different places because they were moved out of loops

#### 2.7.5.2 Q: PRINT FUNCTION INFORMATION

Using the `-Q` option will print out each function name as it is compiled, and print some statistics about each pass when it finishes.

#### 2.7.5.3 SAVE-TEMPS

The `-save-temps` or `-save-temps=cwd` option instructs the compiler to keep intermediate files after building.

The intermediate files will be placed in the current directory and have a name based on the corresponding source file. Thus, compiling `foo.c` with `-save-temps` would produce `foo.i`, `foo.s` and the `foo.o` object file.

The `-save-temps=obj` form of this option is similar to `-save-temps=cwd`, but if the `-o` option is specified, the temporary files are based on the object file. If the `-o` option is not specified, the `-save-temps=obj` switch behaves like `-save-temps`.

For example:

```
xc8-gcc -save-temps=obj -c foo.c
xc8-gcc -save-temps=obj -c bar.c -o dir/xbar.o
xc8-gcc -save-temps=obj foobar.c -o dir2/yfoobar
```

would create `foo.i`, `foo.s`; `dir/xbar.i`, `dir/xbar.s`; `dir2/yfoobar.i`, `dir2/yfoobar.s`, and `dir2/yfoobar.o`.

### 2.7.6 Options for Controlling Optimization

The options shown in Table 2-14 control compiler optimizations and are described in the sections that follow.

**TABLE 2-14: GENERAL OPTIMIZATION OPTIONS**

| Option | Edition | Builds with |
|---|---|---|
| -O0 | All | No optimizations (default) |
| -O<br>-O1 | All | Optimization level 1 |
| -O2 | PRO only | Optimization level 2 |
| -O3 | PRO only | Optimization level 3 |
| -Og | All | Better debugging |
| -Os | PRO only | Size orientated optimizations |
| -flto | PRO only | The standard link-time optimizer |
| -fwhole-program | PRO only | The whole-program optimizations |

#### 2.7.6.1 O0: LEVEL OPTIMIZATIONS

The -O0 option disables optimization.

Without -O, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

The compiler only allocates variables declared register in registers.

The following options control specific optimizations. The -O2 option turns on all of these optimizations except -funroll-loops, -funroll-all-loops and -fstrict-aliasing.

#### 2.7.6.2 O1: LEVEL OPTIMIZATIONS

The -O or -O1 options request level 1 optimizations.

The optimizations performed when using -O1 take somewhat longer to perform and require much more host memory when processing large functions. The compiler will try to reduce code size and execution time.

The compiler turns on -fthread-jumps and -fdefer-pop. The compiler turns on -fomit-frame-pointer.

#### 2.7.6.3 O2: LEVEL OPTIMIZATIONS

The -O2 option performs nearly all supported optimizations that do not involve a space-speed trade-off.

At this level, the compiler performs nearly all supported optimizations that do not involve a space-speed trade-off. This option turns on all optional optimizations except for loop unrolling (-funroll-loops), function inlining (-finline-functions), and strict aliasing optimizations (-fstrict-aliasing). It also turns on force copy of memory operands (-fforce-mem) and Frame Pointer elimination (-fomit-frame-pointer). As compared to -O, this option increases both compilation time and the performance of the generated code.

#### 2.7.6.4 O3: LEVEL OPTIMIZATIONS

The `-O3` option requests nearly all supported optimizations, even those that might increase program size.

The `-O3` option turns on all optimizations specified by `-O2` and also turns on the `inline-functions` option.

#### 2.7.6.5 OG: BETTER DEBUGGING

The `-Og` option enables optimizations that do not interfere with debugging, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience.

#### 2.7.6.6 OS: LEVEL OPTIMIZATIONS

The `-Os` option requests nearly all supported optimizations that decrease program size.

The `-Os` option enables all `-O2` optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

#### 2.7.6.7 LTO

This `-flto` option runs the standard link-time optimizer.

When this option is used, the compiler adds a special section to the object file to hold the intermediate code. When the object files are linked together, all the function bodies are read from these sections and instantiated as if they had been part of the same translation unit.

To use the link-timer optimizer, specify `-flto` both at compile time and during the final link. For example

```
xc8-cc -c -O1 -flto -mcpu= atmega3250p foo.c
xc8-cc -c -O1 -flto -mcpu= atmega3250p bar.c
xc8-cc -o myprog.elf -flto -O3 -mcpu=atmega3250p foo.o bar.o
```

Another (simpler) way to enable link-time optimization is,

```
xc8-cc -o myprog.elf -flto -O3 -mcpu=atmega3250p foo.c bar.c
```

Link time optimizations should not be used with the `-fwhole-program` optimizations.

#### 2.7.6.8 WHOLE-PROGRAM

This `-fwhole-program` option runs more aggressive interprocedural optimizations.

When this option is used, the compiler assumes that the current compilation unit represents the whole program being compiled. All public functions and variables, with the exception of `main()` and those merged by attribute `externally_visible`, are assumed to be `static` and in effect are optimized more aggressively by interprocedural optimizers.

Whole-program optimizations should not be used with the `-flto` optimizations.

### 2.7.7 Options for Controlling the Preprocessor

The options shown in Table 2-15 control the preprocessor and are discussed in the sections that follow.

**TABLE 2-15: PREPROCESSOR OPTIONS**

| Option | Controls |
|---|---|
| -C | Preserve comments |
| -dD | Preserve macro definitions |
| -Dmacro<br>-Dmacro=defn | Define a macro |
| -dM | Output macro definition list |
| -dN | Preserve macro names |
| -fno-show-column | Omit column numbers in diagnostics |
| -H | Print header file name |
| -include file | Include file |
| -iquote | Specify quoted include file search path |
| -M | Generate make rule |
| -MD | Write dependency information to file |
| -MF file | Specify dependency file |
| -MG | Ignore missing header files |
| -MM | Generate make rule for quoted headers |
| -MMD | Generate make rule for user headers |
| -MP | Add phony target for dependency |
| -MQ | Change rule target with quotes |
| -MT target | Change rule target |
| -nostdinc | System directories omitted from header search |
| -P | Don't generate #line directives |
| -trigraphs | Support trigraphs |
| -Umacro | Undefine macros |
| -undef | Do not predefine nonstandard macros |

#### 2.7.7.1 C: PRESERVE COMMENTS

The `-C` option tells the preprocessor not to discard comments from the output. Use this option with the `-E` option to see commented yet preprocessed source code.

#### 2.7.7.2 DD: PRESERVE MACRO DEFINITIONS

The `-dD` option is similar to the `-dM` option, but tells the preprocessor to not remove macro definitions from the output.

#### 2.7.7.3 D: DEFINE A MACRO

The `-D`*`macro`* option defines macro called *`macro`* with the text 1 as its definition.

The `-D`*`macro=text`* form of this option defines the macro *`macro`* which will subsequently expand to be the text *`text`*. All instances of `-D` on the command line are processed before any `-U` options.

Defining macros as C string literals requires bypassing any interpretation issues in the operating system that is being used. To pass the C string, `"hello world"`, (including the *quote* characters) in the Windows environment, use: `"-DMY_STRING=\\\"hello world\\\""` (you must include the *quote* characters around the entire option, as there is a *space* character in the macro definition). Under Linux or Mac OS X, use: `-DMY_STRING=\"hello\ world\"`.

#### 2.7.7.4 DM: OUTPUT MACRO DEFINITION LIST

The `-dM` option tells the preprocessor to output only a list of the macro definitions that are in effect at the end of preprocessing. Used with the `-E` option.

#### 2.7.7.5 DN: PRESERVE MACRO NAMES

The `-dN` option is like `-dD` except that the macro arguments and contents are omitted. Only the macro name is included in the output.

#### 2.7.7.6 NO-SHOW-COLUMN

The `-fno-show-column` option controls whether column numbers will be printed in diagnostics. This may be necessary if diagnostics are being scanned by a program that does not understand the column numbers, such as DejaGnu.

#### 2.7.7.7 H: PRINT HEADER FILES

The `-H` option prints to the console the name of each header file used, in addition to other normal activities.

#### 2.7.7.8 INCLUDE

The `-include` *`file`* option processes *`file`* as if `#include "file"` appeared as the first line of the primary source file. In effect, the contents of file are compiled first. Any `-D` and `-U` options on the command line are always processed before `-include` file, regardless of the order in which they are written. All the `-include` and `-imacros` options are processed in the order in which they are written.

#### 2.7.7.9 IQUOTE

The `-iquote` *`dir`* option adds the directory *`dir`* to the list of directories to be searched for header files during preprocessing. Directories specified with `-iquote` apply only to the quoted form of the directive, `#include "`*`file`*`"`.

#### 2.7.7.10 M: GENERATE MAKE RULE

The `-M` option tells the preprocessor to output a rule suitable for make describing the dependencies of each object file. For each source file, the preprocessor outputs one make-rule whose target is the object file name for that source file and whose dependencies are all the header files it includes. This rule may be a single line or may be continued with `\-`*`newline`* if it is long. The list of rules is printed on standard output instead of the preprocessed C program.

The `-M` option implies `-E`.

#### 2.7.7.11 MD: WRITE DEPENDENCY INFORMATION TO FILE

The `-MD` option is similar to `-M` but the dependency information is written to a file and compilation continues. The file containing the dependency information is given the same name as the source file with a `.d` extension.

#### 2.7.7.12 MF: SPECIFY DEPENDENCY FILE

The `-MF` *`file`* option specifies a file in which to write the dependencies for the `-M` or `-MM` options. If no `-MF` switch is given, the preprocessor sends the rules to the same place it would have sent preprocessed output.

When used with the driver options, `-MD` or `-MMD`, `-MF`, overrides the default dependency output file.

#### 2.7.7.13 MG: IGNORE MISSING HEADER FILES

The `-MG` option treats missing header files as generated files and adds them to the dependency list without raising an error. It assumes the files live in the same directory as the source file. If `-MG` is specified, then either `-M` or `-MM` must also be specified. The `-MG` option is not supported with `-MD` or `-MMD`.

#### 2.7.7.14 MM: GENERATE MAKE RULE FOR QUOTED HEADERS

The `-MM` option is like `-M` but the output mentions only the user header files included with `#include "file"`. System header files included with `#include <file>` are omitted.

#### 2.7.7.15 MMD: GENERATE MAKE RULE FOR USER HEADERS

The `-MMD` option is like `-MD` except mention only user header files, not system header files.

#### 2.7.7.16 MP: ADD PHONY TARGET FOR DEPENDENCY

The `-MP` option instructs the preprocessor to add a phony target for each dependency other than the main file, causing each to depend on nothing. These dummy rules work around `make` errors if you remove header files without updating the make-file to match.

This is typical output:

```
test.o: test.c test.h
test.h:
```

#### 2.7.7.17 MQ: CHANGE RULE TARGET WITH QUOTES

The `-MQ` option is similar to `-MT`, but it quotes any characters which are special to make.

`-MQ '$(objpfx)foo.o'` gives `$$(objpfx)foo.o: foo.c`

The default target is automatically quoted, as if it were given with `-MQ`.

#### 2.7.7.18 MT: CHANGE RULE TARGET

The `-MT` *`target`* option changes the target of the rule emitted by dependency generation. By default, the preprocessor takes the name of the main input file, including any path, deletes any file suffix such as `.c`, and appends the platform's usual object suffix. The result is the target.

An `-MT` option sets the target to be exactly the string you specify. If you want multiple targets, you can specify them as a single argument to `-MT`, or use multiple `-MT` options.

For example:

`-MT '$(objpfx)foo.o'` might give `$(objpfx)foo.o: foo.c`

#### 2.7.7.19 NOSTDINC

The `-nostdinc` option removes the standard system directories from the search for header files. Only the directories you have specified with `-I` options (and the current directory, if appropriate) are searched.

By using both `-nostdinc` and `-iquote`, the include-file search path can be limited to only those directories explicitly specified.

#### 2.7.7.20 P: DON'T GENERATE #LINE DIRECTIVES

The `-P` option tells the preprocessor not to generate `#line` directives in the preprocessed output. Used with the `-E` option.

#### 2.7.7.21 TRIGRAPHS

The `-trigraphs` option turns on support for ANSI C trigraphs. The `-ansi` option also has this effect.

#### 2.7.7.22 U: UNDEFINE MACROS

The `-U`*`macro`* option undefines the macro *`macro`*. All `-U` options are evaluated after all `-D` options, but before any `-include` and `-imacros` options.

#### 2.7.7.23 UNDEF

The `-undef` option prevents any system-specific or GCC-specific macros being pre-defined (including architecture flags).

### 2.7.8 Options for Assembling

The options shown in Table 2-16 control assembler operations and are discussed in the sections that follow.

**TABLE 2-16: ASSEMBLY OPTIONS**

| Option | Controls |
|---|---|
| `-Wa,`*`option`* | Options to passed to the assembler. |
| `-Xassembler` *`option`* | Options to passed to the assembler. |

#### 2.7.8.1 WA: PASS OPTION TO THE ASSEMBLER

The `-Wa,option` option passes its option argument directly to the assembler. If option contains commas, it is split into multiple options at the commas.

#### 2.7.8.2 XASSEMBLER ASSEMBLER OPTION

The `-Xassembler,`*`option`* option pass *`option`* to the assembler where it will be interpreted as an assembler option. You can use this to supply system-specific assembler options that the compiler does not know how to recognize.

### 2.7.9 Mapped Assembler Options

The option shown in Table 2-17 is a commonly used assembler option.

**TABLE 2-17: MAPPED ASSEMBLER OPTIONS**

| Option | Controls |
|---|---|
| `-Wl,-a` | The generation of an assembly list file |

### 2.7.10 Options for Linking

The options shown in Table 2-18 control linker operations and are discussed in the sections that follow. If any of the options `-c`, `-S` or `-E` are used, the linker is not run.

**TABLE 2-18: LINKING OPTIONS**

| Option | Controls |
|---|---|
| `-l`*library* | Which library files are scanned |
| `-nodefaultlibs` | Whether library code is linked with the project |
| `-nostartfiles` | Whether the runtime startup module is linked in |
| `-nostdlib` | Whether the library and startup code is linked with the project |
| `-s` | Remove all symbol table and relocation information from the executable. |
| `-u` *symbol* | The linking in of library modules so that *symbol* can be defined. It is legitimate to use `-u` multiple times with different symbols to force loading of additional library modules. |
| `-Wl,`*option* | Options passed to the linker. |
| `-Xlinker` *option* | System-specific options to passed to the linker |

#### 2.7.10.1 LLIBRARY

The `-l`*library* option scans the library named *library* for unresolved symbols when linking.

The linker searches a standard list of directories for the library with the name *library.a*.

It makes a difference where you write this option in the command. The linker processes libraries and object files in the order they are specified. Thus, `foo.o -lz bar.o` searches library `z` after file `foo.o` but before `bar.o`. If `bar.o` refers to functions in `libz.a`, those functions may not be loaded.

The directories searched include several standard system directories, plus any that you specify with `-L`.

Normally the files found this way are library files (archive files whose members are object files). The linker handles an archive file by scanning through it for members which define symbols that have been referenced but not defined yet. But if the file found is an ordinary object file, it is linked in the usual fashion. The only difference between using an `-l` option (e.g., `-lmylib`) and specifying a file name (e.g., `mylib.a`) is that the compiler will search for a library specified using `-l` in several directories, as specified by the `-L` option.

#### 2.7.10.2 NODEFAULTLIBS

The `-nodefaultlibs` option will prevent the standard system libraries being used when linking. Only the libraries you specify are passed to the linker.

#### 2.7.10.3 NOSTARTFILES

The `-nostartfiles` option will prevent the runtime startup modules from being linked into the project.

#### 2.7.10.4 NOSTDLIB

The `-nostdlib` option will prevent the standard system start-up files and libraries being used when linking. No start-up files and only the libraries you specify are passed to the linker.

#### 2.7.10.5 S: REMOVE SYMBOL INFORMATION

The `-s` option removes all symbol table and relocation information from the output.

#### 2.7.10.6 U: ADD UNDEFINED SYMBOL

The `-u` option adds an undefined symbol at the link stage. To resolve the symbol, the linker will search library modules for its definition, thus this option is useful if you want to force a library module to be linked in. It is legitimate to use `-u` multiple times with different symbols to force loading of additional library modules.

#### 2.7.10.7 WL OPTION

The `-Wl,`*`option`* option pass *`option`* to the linker where it will be interpreted as a linker option. If option contains commas, it is split into multiple options at the commas.

#### 2.7.10.8 XLINKER OPTION

The `-Xl,`*`option`* option pass *`option`* to the linker where it will be interpreted as a linker option. You can use this to supply system-specific linker options that the compiler does not know how to recognize.

### 2.7.11 Mapped Linker Options

The options shown in Table 2-19 are commonly used linker options.

**TABLE 2-19: MAPPED LINKER OPTIONS**

| Option | Controls |
|---|---|
| `-Wl,-[no-]data-init` | Clearing and initialization of C objects at runtime startup |
| `-Wl,-Map=`*`mapfile`* | The generation of a linker map file |

### 2.7.12 Options for Directory Search

The options shown in Table 2-20 control directories searched operations and are discussed in the sections that follow.

**TABLE 2-20: DIRECTORY SEARCH OPTIONS**

| Option | Controls |
|---|---|
| `-idirafter` *`dir`* | Additional directories searched for headers after searching system paths |
| `-imacros` *`file`* | Include file macro definitions only |
| `-I`*`dir`* | The directories searched for preprocessor include files |
| `-L`*`dir`* | The directories searched for libraries |
| `-nostdinc` | The directories searched for headers |

#### 2.7.12.1 IDIRAFTER

The `-idirafter` *`dir`* option adds the directory *`dir`* to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path, including those specified by `-I`.

#### 2.7.12.2 IMACRO

The `-imacros file` option processes *file* exactly like when using the `-include` option, except that any output produced by scanning the file is thrown away. The Macros it defines remain defined. Because the output generated from the file is discarded, the only effect of `-imacros` *file* is to make the macros defined in file available for use in the main input.

Any `-D` and `-U` options on the command line are always processed before `-imacros` *file*, regardless of the order in which they are written. All the `-include` and `-imacros` options are processed in the order in which they are written.

#### 2.7.12.3 I: SPECIFY INCLUDE FILE SEARCH PATH

The `-I` *dir* option adds the directory *dir* to the head of the list of directories to be searched for header files.

This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one `-I` option, the directories are scanned in left-to-right order. The standard system directories come after.

The option can specify either an absolute or relative path, and it can be used more than once if multiple additional directories are to be searched, in which case they are scanned from left to right.The standard system directories are searched after scanning the directories specified with this option.

Under the Windows OS, the use of the directory backslash character may unintentionally form an escape sequence. To specify an include file path that ends with a directory separator character and which is quoted, use `-I "E:\\"`, for example, instead of `-I "E:\"`, to avoid the escape sequence `\"`. Note that MPLAB X IDE will quote any include file path you specify in the project properties and that search paths are relative to the output directory, not the project directory.

#### 2.7.12.4 LDIR

The `-L`*dir* option adds the directory *dir* to the list of directories to be searched for libraries specified by the command line option `-l`.

#### 2.7.12.5 NOSTDINC

The `-nostdinc` option prevents the standard system directories for header files being searched by the preprocessor. Only the directories you have specified with `-I` options and the current directory (if appropriate) are searched.

### 2.7.13 Options for Code Generation Conventions

The options shown in Table 2-21 control machine-independent conventions used when generating code and are discussed in the sections that follow.

**TABLE 2-21: CODE GENERATION CONVENTION OPTIONS**

| Option | Controls |
|---|---|
| `-fshort-enums` | The size of `enum` types |

#### 2.7.13.1 -FSHORT-ENUMS

The `-fshort-enums` option allocates the smallest possible integer type to an enum such that the range of possible values can be held. Use of this option generates code that is not binary compatible with code generated without the option.

# Chapter 3. C Language Features

## 3.1 INTRODUCTION

The MPLAB XC8 C Compiler supports a number of special features and extensions to the C language which are designed to ease the task of producing ROM-based applications for 8-bit AVR devices. This chapter documents the special language features which are specific to these devices.

- C Standard Compliance
- Device-Related Features
- Supported Data Types and Variables
- Memory Allocation and Access
- Operators and Statements
- Register Usage
- Functions
- Interrupts
- Main, Runtime Startup and Reset
- Libraries
- Mixing C and Assembly Code
- Optimizations
- Preprocessing
- Linking Programs

## 3.2 C STANDARD COMPLIANCE

This compiler is a freestanding implementation that conforms to the ISO/IEC 9899:1990 Standard (referred to as the C90 standard) as well the ISO/IEC 9899:1999 Standard (C99) for programming languages. The program standard can be selected using the `-std` option (see Section 2.7.3.8 "std").

This implementation make no assumptions about any underlying operating system, and does not provide support for streams, files, or threads. Aspects of the compiler that diverge from the standards are discussed in this section.

### 3.2.1 Common C Interface Standard

This compiler conforms to the Microchip XC compiler Common C Interface standard (CCI), and can verify that C source code is compliant with CCI.

CCI is a further refinement of the C standards that attempts to standardize implementation-defined behavior and non-standard extensions across the entire MPLAB XC compiler family.

CCI can be enforced by using the `-mext=cci` option (see Section 2.7.3.7 "ext").

### 3.2.2 Divergence from the C99 Standard

The C language implemented on MPLAB XC8 C Compiler can diverge from the C99 Standard in several areas.

#### 3.2.2.1 COMPLEX NUMBER SUPPORT

The complex type `_Imaginary` is not supported (although the use of `_Complex` is permitted). The `<complex.h>` header is also not supported.

## 3.3 DEVICE-RELATED FEATURES

MPLAB XC8 has several features which relate directly to the 8-bit AVR architectures and instruction sets. These are detailed in the following sections.

### 3.3.1 Device Support

The MPLAB XC8 C Compiler aims to support all 8-bit PIC and AVR devices (excluding only the avr1 architecture devices, which must be programmed in assembly). This user's guide should be consulted when you are programming an 8-bit AVR device; when programming a PIC target, see the MPLAB® XC8 C Compiler User's Guide for PIC® MCU (DS50002737).

### 3.3.2 Instruction Set Support

The compiler support all instruction sets for all 8-bit AVR devices, excluding that for the avr1 architecture.

### 3.3.3 Device Header Files

There is one header file that is typically included into each C source file you write. The file is `<xc.h>` and is a generic header file that will include other device- and architecture-specific header files when you build your project. This file can also be included into assembly source files.

Inclusion of this file will allow access to special function registers, as well as device-specific macros definitions.

The header files shipped with the compiler are specific to that compiler version. Future compiler versions may ship with modified header files. Avoid including header files that have been copied into you project. Such projects might no longer be compatible with future versions of the compiler.

### 3.3.4 Stacks

There is one stack implemented by MPLAB XC8. This stack is used for both function return addresses and stack-based objects allocated by functions. The registers r28 and r29 (Y pointer) act as a frame pointer from which stack-based objects can be accessed.

The stack pointer is initialized to the highest valid data memory address. As functions are called, they allocate a chunk of memory for the stack-based objects and the stack grows down in memory, towards smaller addresses. When the function exits, the memory it claimed is made available to other functions.

Note that the compiler cannot detect for overflow of the memory reserved for the stack as a whole. There is no runtime check made for software stack overflows. If the software stack overflows, data corruption and code failure might result.

### 3.3.5 Configuration Bit Access

Configuration bits or fuses are used to set up fundamental device operation, such as the oscillator mode, watchdog timer, programming mode and code protection. These bits must be correctly set to ensure your program executes correctly.

Use the configuration pragma, which has the following form, to set up your device.

```
#pragma config setting = state|value
```

Here, *setting* is a configuration setting descriptor, e.g., WDT, and *state* is a textual description of the desired state, e.g., SET.

Consider the following examples.

```
#pragma config WDTON = SET
#pragma config EESAVE = CLEAR
#pragma config BODLEVEL = BODLEVEL_4V3
```

One pragma can be used to program several settings by separating each setting-value pair with a comma. For example, the above could be specified with one pragma, as in the following.

```
#pragma config WDTON=SET, EESAVE=CLEAR, BODLEVEL=BODLEVEL_4V3
```

The *value* field is a constant that can be used in preference to a descriptor, as in the following.

```
#pragma config SUT_CKSEL = 0x10
```

Setting-value pairs are not scanned by the preprocessor and they are not subject to macro substitution. The setting-value pairs must not be placed in quotes.

The config pragma does not produce executable code, and ideally it should be placed outside function definitions.

Those bits not specified by a pragma are assigned a default value. Rather than rely on this default value, all the bits in the Configuration Words should be programmed to prevent erratic program behavior.

### 3.3.6 Signatures

A signature value can be used by programming software to verify the program was built for the intended device before it is programmed.

Signatures are specified with each device and can be added to your program by simply including the <avr/signature.h> header file. This header will declare a constant unsigned char array in your code and initialize it with the three signature bytes, MSB first, that are defined in the device's I/O header file. This array is then placed in the .signature section in the resulting linked ELF file. This header file should only be included once in an application.

The three signature bytes used to initialize the array are these defined macros in the device I/O header file, from MSB to LSB: SIGNATURE_2, SIGNATURE_1, SIGNATURE_0.

### 3.3.7 Using SFRs From C Code

The Special Function Registers (SFRs) are memory mapped registers that can be accessed from C programs. Each register can be accessed using a macro that is available once you include <xc.h>. For example:

```
#include <xc.h>
if(EEDR == 0x0)
    PORTA = 0x55;
```

Bits within SFRs can be accessed via a special macro, _BV(), and other macros which represent the bit you wish to access. For example, to set bit #1 in PORTB, use the following.

```
PORTB |= _BV(PB1);
```

To clear both bits #4 and #5 in EECR, use the following.

```
EECR &= ~(_BV(EEPM4) | _BV(EEPM5));
```

In both these examples, the compiler will use the device's single bit set and clear instructions whenever possible.

#### 3.3.7.1 SPECIAL REGISTER ISSUES

Some of the timer-related 16-bit registers internally use an 8-bit wide temporary register (called TEMP in the device data sheets) to guarantee atomic access to the timer, since two separate byte transfers are required to move timer values. Typically, this register is used by the device when accessing the current timer/counter value register (TCNT*n*), the input capture register (ICR*n*), and when writing the output compare registers (OCR*n*M). Refer to your device data sheet to determine which peripherals make use of the TEMP register.

This temporary register is not accessible to your program, but it is shared by many peripherals, thus your program needs to ensure that the register is not corrupted by interrupt routines that also uses this register.

Within main-line code, interrupts could be disabled during the execution of the code which utilizes this register. That can be done be encapsulating the code in calls to the cli() and sei() macros, but if the status of the global interrupt flag is not known, the following example code can be used.

```
unsigned int read_timer1(void)
{
    unsigned char sreg;
    unsigned int val;

    sreg = SREG;    // save state of interrupt
    cli();          // disbale interrupts
    val = TCNT1;    // read timer value register; TEMP used internally
    SREG = sreg;    // restore state of interrupts

    return val;
}
```

## 3.4 SUPPORTED DATA TYPES AND VARIABLES

### 3.4.1 Identifiers

Identifiers used to represent C objects and functions must conform to strict rules.

A C identifier is a sequence of letters and digits, where the underscore character “_” counts as a letter. Identifiers cannot start with a digit. Although they can start with an underscore, such identifiers are reserved for the compiler’s use and should not be defined by your programs.

Identifiers are case sensitive, so `main` is different to `Main`.

### 3.4.2 Integer Data Types

The MPLAB XC8 compiler supports integer data types with 1, 2, 4 and 8 byte sizes. Table 3-1 shows the data types and their corresponding size and arithmetic type.

**TABLE 3-1: INTEGER DATA TYPES**

| Type | Size (bits) | Arithmetic Type |
|---|---|---|
| `signed char` | 8 | Signed integer |
| `unsigned char` | 8 | Unsigned integer |
| `signed short` | 16 | Signed integer |
| `unsigned short` | 16 | Unsigned integer |
| `signed int` | 16 | Signed integer |
| `unsigned int` | 16 | Unsigned integer |
| `signed long` | 32 | Signed integer |
| `unsigned long` | 32 | Unsigned integer |
| `signed long long` | 64 | Signed integer |
| `unsigned long long` | 64 | Unsigned integer |

If no type signedness is specified (even for `char` types), then that type is `signed`.

All integer values are represented in little endian format with the Least Significant Byte (LSB) at the lower address.

Signed values are stored as a two’s complement integer value.

The range of values capable of being held by these types is summarized in Table A-8 The symbols in this table are preprocessor macros which are available after including `<limits.h>` in your source code. As the size of data types are not fully specified by the C Standard, these macros allow for more portable code which can check the limits of the range of values held by the type on this implementation.

Macros are also available in `<stdint.h>` which define values associated with fixed-width types, such as `int8_t`, `uint32_t` etc.

### 3.4.3 Boolean Types

The compiler supports `_Bool`, a type used for holding true and false values. The values held by variables of this type are not integers. Values converted to a `_Bool` type result in 0 (false) if the value is 0; otherwise, they result in 1 (true).

The `<stdbool.h>` header defines `true` and `false` macros that can be used with `_Bool` types, and the `bool` macro, which expands to the `_Bool` type. For example:

```
#include <stdbool.h>
_Bool motorOn;
motorOn == false;
```

### 3.4.4 Floating-Point Data Types

The MPLAB XC8 compiler supports 32-bit floating-point types. Floating point is implemented using a IEEE 754 32-bit format. Table 3-2 shows the data types.

**TABLE 3-2: FLOATING-POINT DATA TYPES**

| Type | Size (bits) | Arithmetic Type |
|---|---|---|
| `float` | 32 | Real |
| `double` | 32 | Real |
| `long double` | 32 | Real |

Floating-point types are always signed and the `unsigned` keyword is illegal when specifying a floating-point type. All floating-point values are represented in little endian format with the LSB at the lower address.

Infinities are legal arguments for all operations and behave as the largest representable number with that sign. For example, +inf + -inf yields the value 0.

The format for both floating-point types is described in Table 3-3, where:

- *Sign* is the sign bit, which indicates whether the number is positive or negative.
- The *Biased Exponent* is 8 bits wide and is stored as excess 127 (i.e., an exponent of 0 is stored as 127).
- *Mantissa*, is located to the right of the radix point. There is an implied bit to the left of the radix point which is always 1 except for a zero value, where the implied bit is zero. A zero value is indicated by a zero exponent.

The value of this number is $(-1)^{sign} \times 2^{(exponent-127)} \times 1.$ mantissa.

**TABLE 3-3: FLOATING-POINT FORMATS**

| Format | Sign | Biased Exponent | Mantissa |
|---|---|---|---|
| IEEE 754 32-bit | x | xxxx xxxx | xxx xxxx xxxx xxxx xxxx xxxx |

An example of the IEEE 754 32-bit format shown in Table 3-4. Note that the Most Significant Bit (MSb) of the mantissa column (i.e., the bit to the left of the radix point) is the implied bit, which is assumed to be 1 unless the exponent is zero.

**TABLE 3-4: FLOATING-POINT FORMAT EXAMPLE IEEE 754**

| Format | Value | Biased Exponent | 1.mantissa | Decimal |
|---|---|---|---|---|
| 32-bit | 7DA6B69Bh | 11111011b | 1.0100110101101101 0011011b | 2.77000e+37 |
| | | (251) | (1.302447676659) | — |

The sign bit is zero; the biased exponent is 251, so the exponent is 251-127=124. Take the binary number to the right of the decimal point in the mantissa. Convert this to decimal and divide it by 223 where 23 is the size of the mantissa, to give 0.302447676659. Add 1 to this fraction. The floating-point number is then given by:

$-1^{0}x2^{124}x1.302447676659$

which is approximately equal to:

2.77000e+37

Binary floating-point values are sometimes misunderstood. It is important to remember that not every floating-point value can be represented by a finite sized floating-point number. The size of the exponent in the number dictates the range of values that the number can hold and the size of the mantissa relates to the spacing of each value that can be represented exactly.

For example, if you are using a 32-bit wide floating-point type, it can store the value 95000.0 exactly. However, the next highest value which can be represented is (approximately) 95000.00781.

The characteristics of the floating-point formats are summarized in Table A-5, where *XXX* can be either `FLT` or `DBL`, representing `float` and `double` types, respectively.

The symbols in this table are preprocessor macros that are available after including `<float.h>` in your source code. As the size and format of floating-point data types are not fully specified by the C Standard, these macros allow for more portable code which can check the limits of the range of values held by the type on this implementation.

### 3.4.5 Structures and Unions

MPLAB XC8 C Compiler supports `struct` and `union` types. Structures and unions only differ in the memory offset applied to each member.

These types will be at least 1 byte wide. Bit-fields and `_Bool` types are fully supported.

Structures and unions can be passed freely as function arguments and function return values. Pointers to structures and unions are fully supported.

#### 3.4.5.1 STRUCTURE AND UNION QUALIFIERS

The compiler supports the use of type qualifiers on structures. When a qualifier is applied to a structure, all of its members will inherit this qualification. In the following example the structure is qualified `const`.

```
const struct {
        int number;
        int *ptr;
} record = { 0x55, &i };
```

In this case, each structure member will be read-only. Remember that all members should be initialized if a structure is `const`, as they cannot be initialized at runtime.

#### 3.4.5.2 BIT-FIELDS IN STRUCTURES

MPLAB XC8 C Compiler fully supports bit-fields in structures.

Bit-fields are always allocated within 8-bit words, even though it is usual to use the type `unsigned int` in the definition.

The first bit defined will be the LSb of the word in which it will be stored. When a bit-field is declared, it is allocated within the current 8-bit unit if it will fit; otherwise, a new byte is allocated within the structure. Bit-fields can span the boundary between 8-bit allocation units; however, the code to access bit-fields that do so is extremely inefficient.

For example, the declaration:

```
struct {
        unsigned        lo : 1;
        unsigned        dummy : 6;
        unsigned        hi : 1;
} foo;
```

will produce a structure occupying 1 byte. If `foo` was ultimately linked at address 0x100, the field `lo` will be bit 0 of address 0x100 and field `hi` will be bit 7 of the same address 0x100. The LSb of `dummy` will be bit 1and the MSb of `dummy` will be bit 6.

> **Note:** Accessing bit-fields larger than a single bit can be very inefficient. If code size and execution speed are critical, consider using a `char` type or a `char` structure member instead.

Unnamed bit-fields can be declared to pad out unused space between active bits in control registers. You might use this feature to ensure that bit-fields are wholly contained within a byte. For example, if `dummy` is never referenced, the structure above could have been declared as:

```
struct {
        unsigned        lo : 6;
        unsigned           : 2;
        unsigned        hi : 4;
} foo;
```

A structure with bit-fields can be initialized by supplying a *comma*-separated list of initial values for each field. For example:

```
struct {
        unsigned        lo  : 1;
        unsigned        mid : 6;
        unsigned        hi  : 1;
} foo = {1, 8, 0};
```

Structures with unnamed bit-fields can be initialized. No initial value should be supplied for the unnamed members, for example:

```
struct {
        unsigned        lo  : 1;
        unsigned            : 6;
        unsigned        hi  : 1;
} foo = {1, 0};
```

will initialize the members `lo` and `hi` correctly.

A bit-field that has a size of 0 is a special case. The Standard indicates that no further bit-field is to be packed into the allocation unit in which the previous bit-field, if any, was placed.

#### 3.4.5.3 ANONYMOUS STRUCTURES AND UNIONS

The MPLAB XC8 compiler supports anonymous structures and unions. These are constructs with no identifier and whose members can be accessed without referencing the identifier of the construct. Anonymous structures and unions must be placed inside other structures or unions. For example:

```
struct {
       union {
       int x;
       double y;
    };
} aaa;

void main(void)
{
    aaa.x = 99;
    // ...}
```

Here, the union is not named and its members accessed as if they are part of the structure.

### 3.4.6 Pointer Types

There are two basic pointer types supported by the MPLAB XC8 C Compiler:

- Data pointers - hold the addresses of objects which can be read (and possibly written) by the program.
- Function pointers - hold the address of an executable function which can be called via the pointer.

Data pointers (even generic `void *` pointers) should never be used to hold the address of functions, and function pointers should never be used to hold the address of objects.

#### 3.4.6.1 COMBINING TYPE QUALIFIERS AND POINTERS

It is helpful to first review the C conventions for definitions of pointer types.

Pointers can be qualified like any other C object, but care must be taken when doing so as there are two quantities associated with pointers. The first is the actual pointer itself, which is treated like any ordinary C variable and has memory reserved for it. The second is the target (or targets) that the pointer references, or to which the pointer points. The general form of a pointer definition looks like the following:

*target_type_&_qualifiers * pointer's_qualifiers pointer's_name;*

Any qualifiers to the right of the `*` (i.e., next to the pointer's name) relate to the pointer variable itself. The type and any qualifiers to the left of the `*` relate to the pointer's targets. This makes sense since it is also the * operator that dereferences a pointer, which allows you to get from the pointer variable to its current target.

Here are three examples of pointer definitions using the `volatile` qualifier. The fields in the definitions have been highlighted with spacing:

```
volatile int *          vip ;
int          * volatile  ivp ;
volatile int * volatile  vivp ;
```

The first example is a pointer called `vip`. The pointer itself – the variable that holds the address – is *not* `volatile`; however, the objects that are accessed when the pointer is dereferenced are treated as being `volatile`. In other words, the target objects accessible via the pointer can be externally modified.

In the second example, the pointer called `ivp` is `volatile`, that is, the address the pointer contains can be externally modified; however, the objects that can be accessed when dereferencing the pointer are not `volatile`.

The last example is of a pointer called `vivp` which is itself qualified `volatile`, and which also holds the address of `volatile` objects.

Bear in mind that one pointer can be assigned the addresses of many objects; for example, a pointer that is a parameter to a function is assigned a new object address every time the function is called. The definition of the pointer must be valid for every target address assigned.

**Note:** Care must be taken when describing pointers. Is a "const pointer" a pointer that points to `const` objects, or a pointer that is `const` itself? You can talk about "pointers to const" and "const pointers" to help clarify the definition, but such terms might not be universally understood.

#### 3.4.6.2 DATA POINTERS

Pointers to objects in the data space are 2 bytes wide.

##### 3.4.6.2.1 Pointers to Both Memory Spaces

When a data pointer is assigned the address of one or more objects that have been allocated memory in the data space, and also assigned the address of one or more objects that have been allocated memory in the program memory space, the pointer is said to have targets with mixed memory spaces.

The `__memx` pointer target qualifier will allow the pointer to be able to read from both data and program memories. This qualifier needs to be used in conjunction with the `const` specifier and the pointer will be larger in size than a regular data pointer, for example the following function can read an `int` from either memory space:

```
int read(const __memx int * mip) {
  return *mip;
}
```

#### 3.4.6.3 FUNCTION POINTERS

The MPLAB XC8 compiler fully supports pointers to functions. These are often used to call one of several function addresses stored in a user-defined C array, which acts like a lookup table.

Function pointers are two bytes in size. As the address is word aligned, such pointers can reach program memory addresses up to 128kB. If the device you are using supports more than this amount of program memory and you wish to indirectly access routines above this address, then you need to use the `-mrelax` option (see Section 2.7.1.6 "relax"), which maintain the size of the pointer, but will instruct the linker to have calls reach their final destination via lookups.

In order to facilitate indirect jump on devices with more than 128 Ki bytes of program memory space, there is a special function register called EIND that serves as most significant part of the target address when `eicall` or `eijmp` instructions are executed. The compiler might also use this register in order to emulate an indirect call or jump by means of a `ret` instruction.

The compiler never sets the EIND register and assumes that it never changes during the startup code or program execution, and this implies that the EIND register is not saved or restored in function or interrupt service routine prologues or epilogues.

To accommodate indirect calls to functions and computed gotos, the linker generates function stubs, or trampolines, that contain direct jumps to the desired addresses. Indirect calls and jumps are made to the stub, which then redirects execution to the desired function or location.

For the stubs to work correctly, the `-mrelax` option must be used. This option ensures that the linker will use a 16-bit function pointer and stub combination, even though the destination address might be above 128 kB.

The default linker script assumes code requires the EIND register contain zero. If this is not the case, a customized linker script must be used in order to place the sections whose name begin with `.trampolines` into the segment appropriate to the value held by the EIND register.

The startup code from the `libgcc.a` library never sets the EIND register.

It is legitimate for user-specific startup code to set up EIND early, for example by means of initialization code located in section `.init3`. Such code runs prior to general startup code that initializes RAM and calls constructors, but after the AVR-LibC startup code that sets the EIND register to a value appropriate for the location of the vector table.

```
#include <avr/io.h>

static void
__attribute__((section(".init3"),naked,used,no_instrument_function))
init3_set_eind (void)
{
  __asm volatile ("ldi r24,pm_hh8(__trampolines_start)\n\t"
                  "out %i0,r24" :: "n" (&EIND) : "r24","memory");
}
```

The `__trampolines_start` symbol is defined in the linker script.

Stubs are generated automatically by the linker, if the following two conditions are met:

- The address of a label is taken by means of the `gs` assembler modifier (short for generate stubs) like so:

```
LDI r24, lo8(gs(func))
LDI r25, hi8(gs(func))
```

- The final location of that label is in a code segment outside the segment where the stubs are located.

The compiler emits `gs` modifiers for code labels in the following situations:

- When taking the address of a function or code label
- When generating a computed goto
- If the prologue-save function is used (see Section 2.7.1.2 “call-prologues”)
- When generating switch/case dispatch tables (these can be inhibited by specifying the `-fno-jump-tables` option, Section 2.7.1.5 “no-jump-tables”)
- C and C++ constructors/destructors called during startup/shutdown

Jumping to absolute addresses is not supported, as shown in the following example:

```
int main (void)
{
    /* Call function at word address 0x2 */
    return ((int(*)(void)) 0x2)();
}
```

Instead, the function has to be called through a symbol (`func_4` in the following example) so that a stub can be set up:

```
int main (void)
{
    extern int func_4 (void);

    /* Call function at byte address 0x4 */
    return func_4();
}
```

The project should be linked with `-Wl,--defsym,func_4=0x4`. Alternatively, `func_4` can be defined in the linker script.

### 3.4.7 Constant Types and Formats

Constants in C are an immediate value that can be specified in several formats that are assigned a type.

#### 3.4.7.1 INTEGRAL CONSTANTS

The format of integral constants specifies their radix. MPLAB XC8 supports the standard radix specifiers, as well as ones which enables binary constants to be specified in C code.

The formats used to specify the radices are given in Table 3-5. The letters used to specify binary or hexadecimal radices are case insensitive, as are the letters used to specify the hexadecimal digits.

**TABLE 3-5: RADIX FORMATS**

| Radix | Format | Example |
|---|---|---|
| binary | `0b`*number* or `0B`*number* | 0b10011010 |
| octal | `0`*number* | 0763 |
| decimal | *number* | 129 |
| hexadecimal | `0x`*number* or `0X`*number* | 0x2F |

Any integral constant will have a type of `int`, `long int` or `long long int`, so that the type can hold the value without overflow. Constants specified in octal or hexadecimal can also be assigned a type of `unsigned int`, `unsigned long int` or `unsigned long long int` if their signed counterparts are too small to hold the value.

The default types of constants can be changed by the addition of a suffix after the digits; e.g., `23U`, where `U` is the suffix. Table 3-6 shows the possible combination of suffixes and the types that are considered when assigning a type. So, for example, if the suffix `l` is specified and the value is a decimal constant, the compiler will assign the type `long int`, if that type will hold the constant; otherwise, it will assigned `long long int`. If the constant was specified as an octal or hexadecimal constant, then unsigned types are also considered.

**TABLE 3-6: SUFFIXES AND ASSIGNED TYPES**

| Suffix | Decimal | Octal or Hexadecimal |
|---|---|---|
| `u` or `U` | `unsigned int`<br>`unsigned long int`<br>`unsigned long long int` | `unsigned int`<br>`unsigned long int`<br>`unsigned long long int` |
| `l` or `L` | `long int`<br>`long long int` | `long int`<br>`unsigned long int`<br>`long long int`<br>`unsigned long long int` |
| `u` or `U`, and `l` or `L` | `unsigned long int`<br>`unsigned long long int` | `unsigned long int`<br>`unsigned long long int` |
| `ll` or `LL` | `long long int` | `long long int`<br>`unsigned long long int` |
| `u` or `U`, and `ll` or `LL` | `unsigned long long int` | `unsigned long long int` |

Here is an example of code that can fail because the default type assigned to a constant is not appropriate:

```
unsigned long int result;
unsigned char shifter;

shifter = 20;
result = 1 << shifter;
```

The constant `1` (one) will be assigned an `int` type, hence the value 1 shifted left 20 bits will yield the result 0, not 0x100000.

The following uses a suffix to change the type of the constant, hence ensure the shift result has an `unsigned long` type.

```
result = 1UL << shifter;
```

### 3.4.7.2 FLOATING-POINT CONSTANT

Floating-point constants have `double` type unless suffixed by `f` or `F`, in which case it is a `float` constant. The suffixes `l` or `L` specify a `long double` type which is considered an identical type to `double` by MPLAB XC8.

Floating point constants can be specified as decimal digits with a decimal point and/or an exponent, or as hexadecimal digits and a binary exponent, initiated with either `p` or `P`. So for example:

```
myFloat = -123.98E12;
myFloat = 0xFFEp-22;
```

#### 3.4.7.3 CHARACTER AND STRING CONSTANTS

Character constants are enclosed by single quote characters, ', for example 'a'. A character constant has `int` type, although this can be later optimized to a `char` type by the compiler.

To comply with the C standard, the compiler does not support the extended character set in characters or character arrays. Instead, they need to be escaped using the back-slash character. For example:

```
const char name[] = "Bj\370rk";
printf("%s's Resum\351", name);     \\ prints "Bjørk's Resumé"
```

Multi-byte character constants are not supported by this implementation.

String constants, or string literals, are enclosed by double quote characters. For example, "`hello world`." The type of string constants is `char *`. The characters that make up the string are stored in data memory, as are all objects qualified `const`.

### 3.4.8 Standard Type Qualifiers

The compiler supports the standard qualifiers `const` and `volatile`, as well additional qualifiers that allow programs take advantage of the 8-bit AVR MCU architecture.

#### 3.4.8.1 CONST TYPE QUALIFIER

The `const` type qualifier is used to tell the compiler that an object is read only and will not be modified. If any attempt is made to modify an object declared `const`, the compiler will issue a warning or error.

#### 3.4.8.2 VOLATILE TYPE QUALIFIER

The `volatile` type qualifier indicates to the compiler that an object cannot be guaranteed to retain its value between successive accesses. This information prevents the optimizer from eliminating apparently redundant references to objects declared `volatile` because these references might alter the behavior of the program.

Any SFR which can be modified by hardware or which drives hardware is qualified as `volatile`, and any variables which can be modified by interrupt routines should use this qualifier as well. For example:

```
volatile static unsigned int  TACTL __at(0x800160);
```

The `volatile` qualifier does not guarantee that any access will be atomic, which is often not the case since the 8-bit AVR architecture can typically access 1 byte of data per instruction.

The code produced by the compiler to access `volatile` objects can be different of that to access ordinary variables and typically the code will be longer and slower for `volatile` objects, so only use this qualifier if it is necessary. Failure to use this qualifier when it is required can lead to code failure.

A common use of the `volatile` keyword is to prevent unused some variables being removed. If a non-`volatile` variable is never used, or used in a way that has no effect, then it can be removed before code is generated by the compiler.

A C statement that consists only of a `volatile` variable's name will produce code that reads the variable's memory location and discards the result. For example, the entire statement, `PORTB;` will produce assembly code the reads `PORTB`.

### 3.4.9 Special Type Qualifiers

The MPLAB XC8 C Compiler supports special type qualifiers to allow the user to control placement of objects with static storage duration into particular address spaces.

#### 3.4.9.1 __MEMX ADDRESS SPACE QUALIFIER

The `__memx` qualifier indicates that the object or the object referenced in the case of pointers is to be accessed via a 24-bit address that can access both program and data memory. This qualifier is not needed when compiling for any device with no separate address spaces, such as the `ATtiny40` that maps the program memory into its data memory space.

This address linearizes flash and RAM, using the high bit of the address to determine which memory space is being accessed. If the MSb is set, the object is accessed from data memory using the lower two bytes as address. If the MSb of the address is clear, data is accessed from program memory, with the `RAMPZ` segment register set according to the high byte of the address.

The function in the following example uses a pointer with this qualifier to indicate that it can be passed the address of objects in any memory space.

```
int readOff(__memx int * ip) { ... }
```

#### 3.4.9.2 __FLASH

The `__flash` qualifier indicates that the object should be located in the program memory. For devices that do not have memory-mapped flash, data is read using the `lpm` instruction.

#### 3.4.9.3 __FLASHN

The `__flash`*n* qualifier, where *n* can range from 1 thru 5, places the object into program memory. For those devices that do not have memory-mapped flash (where applicable), the RAMPZ register is accessed using the `elpm` instruction, allowing access to the full program memory.

### 3.4.10 Attributes

The compiler keyword `__attribute__()` allows you to specify special attributes of objects or structure fields. Place inside the parentheses following this keyword a comma-separated list of the relevant attributes, for example:

```
__attribute__ ((unused))
```

The attribute can be placed anywhere in the object's definition, but is usually placed as in the following example.

```
char __attribute__((weak)) input;
char input __attribute__((weak));
```

> **Note:** It is important to use variable attributes consistently throughout a project. For example, if a variable is defined in file A with the aligned attribute, and declared extern in file B without aligned, then a link error may result.

#### 3.4.10.1 ABSDATA

The `absdata` attribute indicates that the objects can be accessed by the `lds` and `sts` instructions, which take absolute addresses. This attribute is only supported for the reduced AVR Tiny core like ATtiny40.

You must make sure that respective data is located in the address range 0x40-0xbf to prevent out of range errors. One way to achieve this as an appropriate linker description file.

#### 3.4.10.2 ADDRESS

Variables with the `address(addr)` attribute are used to address memory-mapped peripherals that may lie outside the io address range.

```
volatile int porta __attribute__((address (0x600)));
```

To place objects at a specified address in the ordinary data memory, use the `__at()` specifier (see Section 3.5.4 "Absolute Variables").

#### 3.4.10.3 ALIGNED

The `aligned(n)` attributed aligns the object's address with the next *n*-byte boundary, where *n* is an numerical argument to the attribute. If the CCI is enabled (see Section 2.7.3.7 "ext") a more portable macro, `__align(n)` (note the different spelling), is available.

This attribute can also be used on a structure member. Such a member will be aligned to the indicated boundary within the structure.

If the alignment argument is omitted, the alignment of the variable is set to 1 (the largest alignment value for a basic data type).

Note that the aligned attribute is used to increase the alignment of a variable, not reduce it. To decrease the alignment value of a variable, use the `packed` attribute.

#### 3.4.10.4 DEPRECATED

The `deprecated` attribute generates a warning whenever the specified object is used. If an option string argument is present, it will be printed in the warning. If the CCI is enabled (see Section 2.7.3.7 "ext") a more portable macro, `__deprecate` (note the different spelling), is available.

#### 3.4.10.5 IO

Objects defined using the `io(address)` attribute represent memory-mapped peripherals in the I/O space and at the address indicated. Example:

```
volatile int porta __attribute__((io(0x22)));
```

When used without an address, the object it is not assigned an address, but the compiler will still use `in` and `out` instructions where applicable, assuming some other module will assign the object an address. For example:

```
extern volatile int porta __attribute__((io));
```

#### 3.4.10.6 IO_LOW

The `io_low(address)` attribute is similar the `io(address)` attribute, but additionally it informs the compiler that the object lies in the lower half of the I/O area, allowing the use of `cbi`, `sbi`, `sbic` and `sbis` instructions. This attribute also has an `io_low` form, which does not specify an address.

#### 3.4.10.7 PACKED

The `packed` attribute forces the object or structure member to have the smallest possible alignment. If the CCI is enabled (see Section 2.7.3.7 "ext") a more portable macro, `__pack` (note the different spelling), is available.

That is, no alignment padding storage will be allocated for the declaration. Used in combination with the `aligned` attribute, packed can be used to set an arbitrary alignment restriction greater or lesser than the default alignment for the type of the variable or structure member.

#### 3.4.10.8 __PERSISTENT

The `__persistent` attribute is used to indicate that objects should not be cleared by the runtime startup code by having them stored in a different area of memory to other objects. If the CCI is enabled (see Section 2.7.3.7 “ext”) a more portable macro, `__persistent`, is available.

By default, C objects with static storage duration that are not explicitly initialized are cleared on startup. This is consistent with the definition of the C language. However, there are occasions where it is desired for some data to be preserved across a Reset.

For example, the following CCI-compliant code ensures that the variable, `intvar`, is not cleared at startup:

```
void test(void)
{
    static __persistent int intvar;  /* must be static */
    // ...
}
```

#### 3.4.10.9 PROGMEM

The `progmem` attribute can be used to have objects placed in the program memory; however, you can use the more portable `PROGMEM` macro, defined by `<avr/pgmspace.h>`, which maps to this attribute, For example.

```
#include <avr/pgmspace.h>
const unsigned char PROGMEM romChar = 0x55;
```

Note that the object must be qualified as `const` for it to be placed in this read-only area of memory.

#### 3.4.10.10 __SECTION

The `__section(`*`section`*`)` attribute allocates the object to a user-nominated section rather than allowing the compiler to place it in a default section. If the CCI is enabled (see Section 2.7.3.7 “ext”) a more portable macro, `__section(`*`section`*`)`, is available. See Section 3.15.2 “Changing and Linking the Allocated Section” for full information on the use of this qualifier.

For example, the following CCI-compliant code places `foobar` in a unique section called `myData`:

```
int __section("myData") foobar;
```

#### 3.4.10.11 UNUSED

The `unused` attribute indicates to the compiler that the object might not be used and that no warnings should be produced if it is detected as being unused.

#### 3.4.10.12 WEAK

The `weak` attribute causes the declaration to be emitted as a weak symbol. A weak symbol indicates that if a global version of the same symbol is available, that version should be used instead.

When weak is applied to a reference to an external symbol, the symbol is not required for linking. For example:

```
extern int __attribute__((weak)) s;
int foo(void) {
    if (&s)
        return s;
    return 0;      /* possibly some other value */
}
```

In the above program, if `s` is not defined by some other module, the program will still link but `s` will not be given an address. The conditional verifies that `s` has been defined (and returns its value if it has). Otherwise '0' is returned. There are many uses for this feature, mostly to provide generic code that can link with an optional library.

## 3.5 MEMORY ALLOCATION AND ACCESS

Objects you define are automatically allocated to an area of memory that is determined by how and where the object is defined in your program. In some instances, it is possible to alter this allocation. Both these topics are discussed in the following sections.

### 3.5.1 Address Spaces

Most 8-bit AVR devices have a Harvard architecture, which has a separate data memory (RAM) and program memory space. On some devices, the program memory is mapped into and accessible from the data memory space. Some devices also implement EEPROM, which is memory mapped on some devices.

Both the general purpose RAM and SFRs share the same data space; however, SFRs appear in a range of addresses (called the I/O space in the device data sheets) that can be accessed by instructions that access the I/O space, such as the `in` and `out` instructions. If a device has more SFRs than these instructions can address, the registers are located at a higher address and accessed via the `st` and `ld` group of instructions.

The program memory space is primarily for executable code, but data can also be located here. There are several ways the different device families locate and read data from this memory, but all objects located here will be read-only.

### 3.5.2 Objects in Data Space Memory

Most objects are ultimately positioned into the data space memory.

Due to the fundamentally different way in which stack-based (automatic storage duration) an other objects (static storage duration) are allocated memory, they are discussed separately.

> **Note:** The terms "local" and "global" are commonly used to describe variables, but are not ones defined by the language Standard. Variables in the C language are characterized by their storage duration, scope, and linkage.

#### 3.5.2.1 STATIC STORAGE DURATION OBJECTS

Objects which are not allocated space on a stack (all objects excluding `auto`, parameter and `const`-qualified objects) have a static (permanent) storage duration and are located by the compiler into the data memory.

Allocation is performed in two steps. The compiler places each object into a specific section and then the linker places these sections into the relevant memory areas. After placement, the addresses of the objects in those sections can be fully resolved.

The compiler considers three categories of these object, which all relate to the value the object should contain at the time the program begins. Each object category has a corresponding family of sections (see Section 3.15.1 “Compiler-Generated Psects”), which are tabulated below.

| | |
|---|---|
| `bss` | These sections contain any uninitialized objects, which will be cleared by the runtime startup code. |
| `data` | These sections contain the RAM image of initialized objects, whose non-zero value is copied to them by the runtime startup code. |

Section 3.10 “Main, Runtime Startup and Reset” has information on how the runtime startup code operates.

##### 3.5.2.1.1 Static Variables

All `static` objects have static storage duration, even local static objects defined inside a function have a scope limited to that function. Even local static objects can be referenced by a pointer and are guaranteed to retain their value between calls to the function in which they are defined, unless explicitly modified via a pointer.

Objects which are `static` have their initial value assigned only once during the program’s execution. Thus, they can be preferable over initialized `auto` objects which are assigned a value every time the block in they are defined begins execution.

##### 3.5.2.1.2 Object Size Limits

An object with static storage duration cannot be made larger than the available device memory size.

##### 3.5.2.1.3 Changing the Default Allocation

You can change the default memory allocation of objects with static storage duration by either:

- Using specifiers
- Making the objects absolute
- Placing objects in their own section and explicitly linking that section

Variables can be placed in a combined flash and data section by using the `__memx` specifier (see Section 3.4.9.1 “__memx Address Space Qualifier”).

If only a few objects are to be located at specific addresses in data space memory, then those objects can be made absolute (described in Section 3.5.4 “Absolute Variables”). Once variables are made absolute, their address is hard coded in generated output code, they are no longer placed in a section and do not follow the normal memory allocation procedure.

The `.bss` and `.data` sections, in which the different categories of static storage duration objects are allocated, can be shifted as a whole by changing the default linker options. For example, you could move all the persistent variables. See Section 3.15.2 “Changing and Linking the Allocated Section” for more information on changing the default linker options for sections.

Objects can also be placed at specific positions by using the `__section()` specifier (see Section 3.15.2 "Changing and Linking the Allocated Section"), to allocate them to a unique section, then link that section to the required address via an option.

#### 3.5.2.2 AUTOMATIC STORAGE DURATION OBJECTS

Parameters and `auto` objects have automatic storage duration and are allocated space on a software stack[1]. Temporary objects might also be placed on the stack as well. Section 3.3.4 "Stacks" describes the stack used by MPLAB XC8 and the 8-bit AVR devices.

Since objects with automatic storage duration are not in existence for the entire execution of the program, there is the possibility to reclaim memory they use when the objects are not in existence and allocate it to other objects in the program. Typically such objects are stored on some sort of a dynamic data stack where memory can be easily allocated and deallocated by each function. Because this stack is used to create new instances of function objects when the function is called, all functions are reentrant.

The standard `const` qualifier can be used with `auto` objects and these do not affect how they are positioned in memory. This implies that a local `const`-qualified object is still an `auto` object and will be allocated memory in the stack of data space memory.

##### 3.5.2.2.1 Object Size Limits

An object with automatic storage duration cannot be made larger than the stack space available at the time which the object comes into existence. Therefore, the maximum size can vary throughout the program.

##### 3.5.2.2.2 Changing the Default Auto Variable Allocation

All objects with automatic storage duration are located on a stack, thus there is no means to individually move them. They cannot be made absolute nor can they be assigned a unique section.

### 3.5.3 Objects in Program Space

Objects defined using the `progmem` attribute or the `PROGMEM` macro and that have static storage duration are placed in program memory.

The avrtiny and avrxmega3 device families can easily access program-memory objects, since this memory is mapped into the data address space. For other device families, program memory is distinct and is accessed via different code sequences.

#### 3.5.3.1 SIZE LIMITATIONS OF PROGRAM-MEMORY OBJECTS

A program-memory object cannot be made larger than the available device program memory size.

Note that in addition to the data itself, there is also a small amount of code required to access data in program memory for those devices that do not have program memory mapped into the data space. This additional code is included only once, regardless of the size or number of program-memory objects.

1. What is referred to as a software stack in this user's guide is the typical dynamic stack arrangement employed by most computers. It is ordinary data memory accessed by some sort of push and pop instructions, and a stack pointer register.

#### 3.5.3.2 CHANGING THE DEFAULT ALLOCATION

You can change the default memory allocation of objects in program memory by either:

- Making the objects absolute
- Placing objects in their own section and explicitly linking that section

If only a few program-memory objects are to be located at specific addresses in program space memory, then the objects can be made absolute. Absolute variables are described in Section 3.5.4 “Absolute Variables”.

Objects in program memory can also be placed at specific positions by using the `__section()` specifier (see Section 3.15.2 “Changing and Linking the Allocated Section”), to allocate them to a unique section, then link that section to the required address via an option.

### 3.5.4 Absolute Variables

Objects can be located at a specific address when the CCI is enabled (see Section 2.7.3.7 “ext”) by following their declaration with the construct `__at(`*`address`*`)`, where *`address`* is the location in memory where the variable is to be positioned. Such a variable is known as an absolute variable.

Making a variable absolute is the easiest method to place an object at a user-defined location, but it only allows placement at an address which must be known prior to compilation and must be specified for each object to be relocated.

#### 3.5.4.1 ABSOLUTE OBJECTS IN DATA MEMORY

Any object which has static storage duration and which has file scope can be placed at an absolute address in data memory when the CCI is enabled (see Section 2.7.3.7 “ext”). Thus all but `auto` objects can be made absolute.

The address specified for data memory objects must be 0x800000 plus the RAM start address plus the desired offset within the RAM block. So, for example, to place the variable `Portvar` at an offset of 0x60 in the RAM block (data address 0x160) for an ATmega48PB device, use:

```
volatile unsigned char Portvar __at(0x800160);
```

The compiler will mark storage for absolute objects as being used, so that ordinary object will not be assigned these addresses. An error will be issued if there is any overlap of absolute variables with other absolute variables.

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

#### 3.5.4.2 ABSOLUTE OBJECTS IN PROGRAM MEMORY

Any program-memory object which has static storage duration and which has file scope can be placed at an absolute address in program memory when the CCI is enabled (see Section 2.7.3.7 “ext”).

For example:

```
#include <avr/pgmspace.h>
int PROGMEM settings[] __at(0x200) = { 1, 5, 10, 50, 100 };
```

will place the array `settings` at address 0x200 in the program memory.

### 3.5.5 Variables in EEPROM

For devices with on-chip EEPROM, the compiler offers several methods of accessing this memory as described in the following sections.

#### 3.5.5.1 EEPROM VARIABLES

Objects can be placed in the EEPROM by specifying that they be placed in the `.eeprom` section, using the `section` attribute. A warning is produced if the attribute is not supported for the selected device. Check your device data sheet to see the memory available with your device.

The macro, `EEMEM`, is defined in `<avr/eeprom.h>` and can be alternatively used to simplify the definition of objects in EEPROM. For example, both the following definitions create objects which will be stored in EEPROM.

```
int serial __attribute__((section(.eeprom)));
char EEMEM ID[5] = { 1, 2, 3, 4, 5 };
```

Objects in this section are cleared or initialized, as required, just like ordinary RAM-based objects; however, the initialization process is not carried out by the runtime startup code. Initial values are placed into a HEX file and are burnt into the EEPROM when you program the device. If you modify the EEPROM during program execution and then reset the device, these objects will not contain the initial values specified in your code at startup up.

Note that the objects that are in the eeprom section must all use the `const` type qualifier or all not use this qualifier.

#### 3.5.5.2 EEPROM ACCESS FUNCTIONS

You must access objects in EEPROM using special library routines, such as `eeprom_read_byte()` and `eeprom_write_word()`, accessible once you include `<avr/eeprom.h>`.

Code to access EEPROM based objects will be much longer and slower than code to access RAM-based objects. Consider using these routines to copy values from the EEPROM to regular RAM-based objects if you need to use them many times in complex calculations.

### 3.5.6 Variables in Registers

You can define a variable and associate it with a specified register; however, it is generally recommended that register allocation be left to the compiler to achieve optimal results and to avoid code failure.

Register variables are defined by using the `register` keyword and indicating the desired register, as in the following example:

```
register int input asm("r12");
```

A valid AVR device register name must be quoted as an argument to the `asm()`. Such a definition can be placed inside or outside a function, but you cannot made the variable `static`. Support for local register variables is to limited to specifying registers for input and output operands when calling extended in-line assembly.

The compiler reserves the allocated register for the duration of the current compilation unit, but library routines may clobber the register allocated, thus it is recommended that you allocate a register that is normally saved and restored by function calls (a call-saved register, described in Section 3.7 "Register Usage").

You cannot take the address of a register variable.

### 3.5.7 Dynamic Memory Allocation

Dynamic memory allocated from a heap at runtime (by functions like `malloc()` etc) is supported by MPLAB XC8. Given the small amounts of data memory available on the AVR architecture, the allocation scheme is relatively robust.

The memory allocated by dynamic memory functions includes an addition two-byte-wide header that is prepended to the requested memory. This header records the size of the allocation and is used by `free()`. The address returned by the memory allocation functions point to the first usable location that has been allocated. The two bytes located before this address contain the header, so your program should take extra care to ensure these locations are not corrupted.

The implementation maintains a simple freelist that accounts for memory regions that have been returned in previous calls to `free()`. Note that all of this memory is considered to have been successfully added to the heap, so no further checks against stack-heap collisions are done when recycling memory from the freelist.

The freelist itself is not maintained as a separate data structure. The contents of the freed memory regions are written with pointers which link the regions. This requires no additional memory to maintain this list, except for a link pointer, which contains the address of the lowest memory region available for reallocation. Since the size of the region (the two-byte header) and a two-byte link pointer to the next free region are recorded in each region, the minimum region size on the freelist is four bytes.

When allocating memory, the linked memory regions in the freelist are first walked to determine if this contains a memory region that satisfies the request. Regions with the same usable size as that requested are allocated first; otherwise, larger regions are considered, if they are available. If a larger free region has at least four bytes more than that requested, it is split into one region which is returned by the allocation function and the remaining region is left on the free list. If splitting the larger free region would result in a region on the freelist that is less than four bytes in size (i.e. not large enough to hold the header information and link pointer), the larger region is not split and is allocated as a whole.

If no suitable memory region could be found on the freelist, the allocation functions attempt to extend the heap. If the heap is located below the stack, memory will be allocated up to a maximum address of the current stack limit minus `__malloc_margin` bytes, which by default is 32 bytes. If the heap is above the stack, memory will be allocated up to a maximum address of `__malloc_heap_end`.

If no memory can be claimed from the heap, the allocation functions will return `NULL`.

When calling `free()`, a new region will be placed on the freelist. This region will be combined with other contiguous regions, yielding the largest possible entry for further allocations. That way, the heap fragmentation can be minimized. When deallocating the topmost chunk of memory, the size of the heap is reduced.

A call to `realloc()` first determines whether the operation should increase or decrease the size of the existing allocation. If the new request is for a region at least two bytes smaller than the current region, the existing region is split and the region no longer required is passed to the standard `free()` function for insertion into the freelist. If the new request is for a region one byte smaller than the existing region, no operation is performed and the existing region is returned.

When a request to `realloc()` is for a larger memory region, the existing allocation is extended in-place, if possible, without having to copy data to the new region. As a side-effect, this check will also record the size of the largest chunk on the freelist.

If the existing region cannot be extended in-place, but is located at the top of heap with no suitable regions in the freelist, the heap is extended (if possible) without having to copy data to the new region. Otherwise, `malloc()` will be called with the new request size, the data in the existing region will be copied over to the new region, and `free()` will be called on the now defunct region.

The request will fail if the top of the heap would surpass its maximum permissible address.

#### 3.5.7.1 ADJUSTING ALLOCATION FUNCTION BEHAVIOR

There are a number of variables that can be tuned to customize the behavior of functions such as `malloc()`. Any changes to these variables should be made before any memory allocation is made, remembering that library functions might use dynamic memory.

The variables `__malloc_heap_start` and `__malloc_heap_end` can be used to restrict the memory allocated by the `malloc()` function. These variables are statically initialized to point to `__heap_start` and `__heap_end`, respectively, where `__heap_start` is set to an address just beyond the `.bss` section, and `__heap_end` is set to 0, which places the heap is below the stack.

If the heap is located in external RAM, `__malloc_heap_end` must be adjusted accordingly. This can be done either at run-time, by writing directly to this variable, or it can be done automatically at link-time, by adjusting the value of the symbol `__heap_end`.

The following example shows an option that can relocate those input sections mapped to the .data output section in the linker script to location 0x1100 in external RAM. (The option `-Wl,-Tdata=0x801100` could also be used in this situation). The heap will extend up to address 0xffff.

```
-Wl,--section-start,.data=0x801100,--defsym=__heap_end=0x80ffff
```

Since these are addresses in RAM, the MSb is set in the address.

If the heap should be located in external RAM while keeping the ordinary variables in internal RAM, the following options can be used. Note that in this example, there is a 'hole' in memory between the heap and the stack that remain unaccessible by ordinary variables or dynamic memory allocations.

```
-Wl,--defsym=__heap_start=0x802000,--defsym=__heap_end=0x803fff
```

Figure 3. Internal RAM: variables and stack, external RAM: heap

If `__malloc_heap_end` is 0, the memory allocation routines attempt to detect the bottom of the stack in order to prevent a stack-heap collision when extending the heap. They will not allocated memory beyond the current stack limit with a buffer of `__malloc_margin` bytes. Thus, all possible stack frames of interrupt routines that could interrupt the current function, plus all further nested function calls must not require more stack space, or they will risk colliding with the data segment.

The default value of `__malloc_margin` is set to 32.

### 3.5.8 Memory Models

MPLAB XC8 C Compiler does not use fixed memory models to alter allocation of variables to memory. Memory allocation is fully automatic and there are no memory model controls.

## 3.6 OPERATORS AND STATEMENTS

The MPLAB XC8 C Compiler supports all the Standard operators, some of which behave in an implementation defined way, see Appendix B. Implementation-Defined Behavior. The following sections illustrate code operations that are often misunderstood as well as additional operations that the compiler is capable of performing.

### 3.6.1 Integral Promotion

Integral promotion is always applied in accordance with the C standard, but it can confuse those who are not expecting such behavior.

When there is more than one operand to an operator, they typically must be of exactly the same type. The compiler will automatically convert the operands, if necessary, so they have the same type. The conversion is to a “larger” type so there is no loss of information; however, the change in type can cause different code behavior to what is sometimes expected. These form the standard type conversions.

Prior to these type conversions, some operands are unconditionally converted to a larger type, even if both operands to an operator have the same type. This conversion is called *integral promotion*. The compiler performs these integral promotions where required, and there are no options that can control or disable this operation.

Integral promotion is the implicit conversion of enumerated types, `signed` or `unsigned` varieties of `char`, `short int` or bit-field types to either `signed int` or `unsigned int`. If the result of the conversion can be represented by an `signed int`, then that is the destination type, otherwise the conversion is to `unsigned int`.

Consider the following example.

```
unsigned char count, a=0, b=50;
if(a - b < 10)
  count++;
```

The `unsigned char` result of `a - b` is 206 (which is not less than 10), but both `a` and `b` are converted to `signed int` via integral promotion before the subtraction takes place. The result of the subtraction with these data types is -50 (which *is* less than 10) and hence the body of the `if()` statement is executed.

If the result of the subtraction is an `unsigned` quantity, then apply a cast, as in the following example, which forces the comparison to be done as `unsigned int` types:

```
if((unsigned int)(a - b) < 10)
  count++;
```

Another problem that frequently occurs is with the bitwise complement operator, `~`. This operator toggles each bit within a value. Consider the following code.

```
unsigned char count, c;
c = 0x55;
if( ~c == 0xAA)
  count++;
```

If c contains the value 0x55, it often assumed that ~c will produce 0xAA; however, the result is 0xFFAA and so the comparison in the above example would fail. The compiler can be able to issue a mismatched comparison error to this effect in some circumstances. Again, a cast could be used to change this behavior.

The consequence of integral promotion as illustrated above is that operations are not performed with char-type operands, but with int-type operands. However, there are circumstances when the result of an operation is identical regardless of whether the operands are of type char or int. In these cases, the compiler might not perform the integral promotion so as to increase the code efficiency. Consider this example.

```
unsigned char a, b, c;
a = b + c;
```

Strictly speaking, this statement requires that the values of b and c are promoted to unsigned int, the addition is performed, the result of the addition is cast to the type of a, and then that result is assigned. In this case, the value assigned to a will be the same whether the addition is performed as an int or char, and so the compiler can choose to encode the operands using the latter type.

If, in the above example, the type of a was unsigned int, then integral promotion would have to be performed to comply with the C Standard.

### 3.6.2 Rotation

The C language does not specify a rotate operator; however, it does allow shifts. You can follow the C code examples below to perform rotations.

```
unsigned char c;
unsigned int u;
c = (c << 1) | (c >> 7);  // rotate left, one bit
u = (u >> 2) | (u << 14); // rotate right, two bits
```

### 3.6.3 Switch Statements

By default, jump tables can be used to optimize switch() statements. The -fno-jump-tables option prevents these from being used and will use sequences of compare statements instead. Jump tables are usually faster to execute on average, but in particular for switch() statements, where most of the jumps would go to the default label, they might waste a bit of flash memory.

The jump tables use the lpm assembler instruction for access to jump tables. Always use the -fno-jump-tables option when compiling a bootloader for devices with more than 64 kB of program memory.

## 3.7 REGISTER USAGE

The assembly generated from C source code by the compiler will use certain registers in the AVR register set. Some registers are assumed to hold their value over a function call.

The call-used registers, r18-r27 and r30-r31, can be allocated by the compiler for values within a function. Functions do not need to preserve the content of these registers. These registers may be used in hand-written assembler subroutines. Since any C function called by these routines can clobber these registers, the calling routine must ensure they are saved as restored as appropriate.

The call-saved registers, r2-r17 and r28-r29, can also be allocated by the compiler for local data; however, C functions must preserve these registers. Hand-written assembler subroutines are responsible for saving and restoring these registers when necessary. The registers must be saved even when the compiler has assigned them for argument passing.

The temporary register, r0, can be clobbered by C functions, but they are saved by interrupt handlers.

The compiler assumes that the Zero register, r1, always contains the value zero. It can be used in hand-written assembly routine for intermediate values, but must be cleared after use (e.g using `clr r1`). Be aware that multiplication instructions return their result in the r1-r0 register pair. Interrupt handlers save and clear r1 on entry, and restore r1 on exit (in case it was non-zero).

All registers that have been used by an interrupt routine are save and restored by the interrupt routine (see Section 3.9.4 “Context Switching”).

The registers that have a dedicated function throughout the program are listed in Table 3-7.

**TABLE 3-7: REGISTERS WITH DEDICATED USE**

| Register name | Applicable devices |
|---|---|
| `r0` | Temporary register |
| `r1 (__zero_reg__)` | Zero register (holds 0 value) |
| `r28, r29` | Frame pointer (Y pointer) |

## 3.8 FUNCTIONS

Functions are written in the usual way, in accordance with C language. Implementation and special features associated with functions are discussed in following sections.

### 3.8.1 Function Specifiers

Aside from the standard C specifier, `static`, which affects the linkage of the function, there are several non-standard function specifiers, which are described in the following sections.

#### 3.8.1.1 __INTERRUPT SPECIFIER

The `__interrupt()` specifier indicates that the function is an interrupt service routine and that it is to be encoded specially to suit this task. Interrupt functions are described in detail in 3.9.1 Writing an Interrupt Service Routine.

#### 3.8.1.2 INLINE SPECIFIER

The `inline` function specifier is a recommendation that the compiler replace calls to the specified function with the function's body, if possible.

The following is an example of a function which has been made a candidate for inlining.

```
inline int combine(int x, int y) {
    return 2*x-y;
}
```

All function calls to a function that was in-lined by the compiler will be encoded as if the call was replaced with the body of the called function. This is performed at the assembly code level. In-lining will only take place if the optimizers are enabled, but you can ask that a function always be in-lined by using the `always_inline` attribute.

If inlining takes place, this will increase the program's execution speed, since the call and return sequences associated with the call will be eliminated. Code size can be reduced if the assembly code associated with the body of the in-lined function is very small, but code size can increase if the body of the in-lined function is larger than the call/return sequence it replaces. You should only consider this specifier for functions which generate small amounts of assembly code. Note that the amount of C code in the body of a function is not a good indicator of the size of the assembly code that it generates.

There are several reasons why the compiler might not in-line a function which has been specified, so your code should not make any assumption about whether inlining took place. The `-Winline` option can warn you when a function marked in-line could not be substituted, and gives the reason for the failure.

#### 3.8.1.3 __SECTION QUALIFIER

The `__section(`*`section`*`)` qualifier allocates the function to a user-nominated section rather than allowing the compiler to place it in a default section. See Section 3.15.2 "Changing and Linking the Allocated Section" for full information on the use of this qualifier.

### 3.8.2 Function Attributes

#### 3.8.2.1 NAKED

The naked attribute allows the compiler to construct the requisite function declaration, while allowing the body of the function to be assembly code. The specified function will not have prologue or epilogue sequences generated by the compiler. Only basic asm statements can safely be included in naked functions. Do not use extended asm or a mixture of basic asm and C code as they are not supported.

#### 3.8.2.2 OS_MAIN AND OS_TASK

On AVR, functions with the OS_main or OS_task attribute do not save/restore any call-saved register in their prologue/epilogue.

The OS_main attribute can be used when there is guarantee that interrupts are disabled at the time when the function is entered. This saves resources when the stack pointer has to be changed to set up a frame for local variables.

The OS_task attribute can be used when there is no guarantee that interrupts are disabled at that time when the function is entered like for, e.g. task functions in a multi-threading operating system. In that case, changing the stack pointer register is guarded by save/clear/restore of the global interrupt enable flag.

The differences to the naked function attribute are:

- naked functions do not have a return instruction whereas OS_main and OS_task functions have a RET or reti return instruction.
- naked functions do not set up a frame for local variables or a frame pointer whereas OS_main and OS_task do this as needed.

#### 3.8.2.3 WEAK

The weak attribute causes the declaration to be emitted as a weak symbol. A weak symbol indicates that if a global version of the same symbol is available, that version should be used instead. This is useful when a library function is implemented such that it can be overridden by a user written function.

For example:

```
int __attribute__((weak)) process(int control)
{
  // ...
}
```

### 3.8.3 External Functions

Functions that are defined outside the projects C source files (e.g., a function defined in a separate bootloader project or in an assembly module) will require declarations so that the compiler knows how to encode calls to those functions.

A function declaration will look similar to the following example. Note that the extern specifier is optional, but make it clear this is a declaration.

```
extern int clockMode(int);
```

### 3.8.4 Allocation of Executable Code

Code associated with C functions is always placed in the `.text` section, which is linked into the program memory of the target device.

### 3.8.5 Changing the Default Function Allocation

You can change the default memory allocation of functions by either:

- Making functions absolute
- Placing functions in their own section and linking that section

The easiest method to explicitly place individual functions at a known address is to make them absolute by using the `__at(address)` construct in a similar fashion to that used with absolute variables. The CCI must be enabled for this syntax to be accepted (see Section 2.7.3.7 "ext")

The compiler will issue a warning if code associated with an absolute function overlaps with code from other absolute functions. The compiler will not locate code associated with ordinary functions over the top of absolute functions.

The following example of an absolute function will place the function at address 400h:

```
int mach_status(int mode) __at(0x400)
{
  /* function body */
}
```

If this construct is used with interrupt functions, it will only affect the position of the code associated with the interrupt function body. The interrupt context switch code associated with the interrupt vector will not be relocated.

Functions can be allocated to a user-defined psect using the `__section()` specifier (see Section 3.15.2 "Changing and Linking the Allocated Section") so that this new section can then be linked at the required location. This method is the most flexible and allows functions to be placed at a fixed address, after other section, or anywhere in an address range. As with absolute functions, when used with interrupt functions, it will only affect the position of the interrupt function body.

Regardless of *how* a function is located, take care choosing its address. If possible, avoid fragmenting memory and increasing the possibility of linker errors.

### 3.8.6 Function Size Limits

For all devices, the code generated for a regular function is limited only by the available program memory. See 3.8.4 Allocation of Executable Code for more details.

### 3.8.7 Function Parameters

MPLAB XC8 uses a fixed convention to pass arguments to a function. The method used to pass the arguments depends on the size and number of arguments involved.

> **Note:** The names “argument” and “parameter” are often used interchangeably, but typically an argument is the value that is passed to the function and a parameter is the variable defined by the function to store the argument.

Arguments are passed to functions via registers and consume as many register as required to hold the object. However, registers are allocated in pairs, thus there will be at least two registers allocated to each argument, even if the argument is a single byte. This make more efficient use of the AVR instruction set.

The first register pair available is r24-r25 and lower register pairs are considered after that down to register r8. For example, if a function with the prototype:

```
int map(unsigned long a, char b);
```

is called, the argument for the four-byte parameter `a` will be passed in registers r22 thru r25, with r22 holding the least significant byte and r25 holding the most significant byte; and the argument for parameter `b` will be assigned to registers r20 and r21.

If there are further arguments to pass once all the available registers have been assigned, they are passed on the stack.

Arguments to functions with variable argument lists (`printf()` etc.) are all passed on stack.

### 3.8.8 Function Return Values

A function’s return value is usually returned in a register.

A byte-sized return value is returned in r24. Multi-byte return values are return in as many registers as required, with the highest register being r25. Thus, a 16-bit value is returned in r24-r25, a 32-bit value in r22-r25, etc.

### 3.8.9 Calling Functions

Functions are called using an `rcall` instruction. If your target device has more than 8kB of program memory, it will use a larger call instruction to be able to reach any function, regardless of where it is located in program memory.

If you can guarantee that all call destinations are within range of the `rcall` instruction, the shorter form of call can be requested by using the `-mshort-calls` option (see Section 2.7.1.7 “short-calls”).

## 3.9 INTERRUPTS

The MPLAB XC8 compiler incorporates features allowing interrupts to be fully handled from C code. Interrupt functions are often called Interrupt Service Routines, or ISRs.

The following are the general steps you need to follow to use interrupts. More detail about these steps is provided in the sections that follow.

- Write as many interrupt functions as required. Consider one or more additional functions to handle accidental triggering of unused interrupt sources.
- At the appropriate point in your code, enable the interrupt sources required.
- At the appropriate point in your code, enable the global interrupt flag.

Interrupt functions must not be called directly from C code (due to the different return instruction used), but can call other functions, such as user-defined and library functions.

*Interrupt code* is the name given to any code that executes as a result of an interrupt occurring, including functions called from the ISR and library code. Interrupt code completes at the point where the corresponding return from interrupt instruction is executed. This contrasts with *main-line code*, which, for a freestanding application, is usually the main part of the program that executes after Reset.

### 3.9.1 Writing an Interrupt Service Routine

Observe the following guidelines when writing an ordinary ISR.

- Write each ISR prototype using the `__interrupt()` specifier. This will create a function with the appropriate name, prototype, and attributes.
- If necessary, clear the relevant interrupt flag once the source has been processed, although typically this is not required.
- Only re-enable interrupts inside the ISR body if absolutely necessary. Interrupt are re-enabled automatically when the ISR returns.
- Keep the ISR as short and as simple as possible. Complex code will typically use more registers that will increase the size of the context switch code.

The compiler will process interrupt functions differently to other functions, generating code to save and restore any registers used by the function and using a special instruction to return.

The hardware globally disables interrupts when an interrupt is executed.

Usually, each interrupt source has a corresponding interrupt flag bit, accessible in a control register. When set, these flags indicate that the specified interrupt condition has been met. Interrupt flags are often cleared in the course of processing the interrupt, either when the handler is invoked or by reading a particular hardware register; however, there are a few instances when the flag must be cleared manually by code. Failure to do so might result in the interrupt triggering again as soon as the current ISR returns.

The flag bits in the SFRs have a unique property whereby they are cleared by writing a logic one to them. To take advantage of this property, you should write directly to the register rather than use any instruction sequence that might perform a read-modify-write. Thus to clear the TOV0 timer overflow flag in the TC0 interrupt flag register, use the following code:

```
TIFR = _BV(TOV0);
```

which is guaranteed to clear the TOV0 bit and leave the remaining bits untouched.

An example of an interrupt function is shown below.

```
void __interrupt(SPI_STC_vect_num) spi_Isr(void) {
    process(SPI_SlaveReceive());
    return;
}
```

Note that the argument to the `__interrupt()` specifier is a vector number, which are available as macros ending with `vect_num` once you have included `<xc.h>` in your program.

More complex interrupt function definitions can be created using macros and attributes defined by `<avr/interrupt.h>` and which are shown in the following examples.

If there is no code to be executed for an interrupt source but you want to ensure that the program will continue normal operation should the interrupt unexpectedly trigger, then you can create an empty ISR using the `EMPTY_INTERRUPT()` macro and an interrupt source argument.

```
#include <avr/interrupt.h>
EMPTY_INTERRUPT(INT2_vect);
```

The special interrupt source symbol, `BADISR_vect`, can be used to define a function that can process any otherwise undefined interrupts. Without this function defined, an undefined interrupt will trigger a device reset. For example:

```
void ISR(BADISR_vect) {
    // place code to process undefined interrupts here
    return;
}
```

If you wish to allow nested interrupts you can manually add an in-line `sei` instruction to your ISR to re-enable the global interrupt flag; however, there is an argument you can use with the `ISR()` macro to have this instruction added by the compiler to the beginning of the interrupt routine. For example:

```
void ISR(IO_PINS_vect, ISR_NOBLOCK)
{ ... }
```

If one ISR is to be used with more than one interrupt vector, then you can define that ISR in the usual way for one vector then reuse that ISR for other vector definitions using the `ISR_ALIASOF()` argument.

```
void __interrupt(PCINT0_vect_num)
{ ... }

void ISR(PCINT1_vect, ISR_ALIASOF(PCINT0_vect));
```

In some circumstances, the compiler-generated context switch code executed by the ISR might not be optimal. In such situations, you can request that the compiler omit this context switch code and supply this yourself. This can be done using the `ISR_NAKED` argument, as shown in this example.

```
void ISR(TIMER1_OVF_vect, ISR_NAKED)
{
  PORTB |= _BV(0);  // results in SBI which does not affect SREG
  reti();
}
```

Note that the compiler will not generate any context switch code, including the final return from interrupt instruction, so you must write any relevant switching code and the `reti` instruction. The SREG register must be manually saved if it is modified by the ISR, and the compiler-implied assumption of `__zero_reg__` always being 0 could be wrong, for example when an interrupt occurs right after a `mul` instruction.

### 3.9.2 Changing the Default Interrupt Function Allocation

You can use the `__at()` specifier (see Section 3.8.5 "Changing the Default Function Allocation") if you want to move the interrupt function itself. This does not alter the position of the vector table, but the appropriate table entry will still point to the correct address of the shifted function.

### 3.9.3 Specifying the Interrupt Vector

The process of populating the interrupt vector locations is fully automatic, provided you define interrupt functions (as shown in Section 3.9.1 "Writing an Interrupt Service Routine"). The compiler will automatically link each ISR entry point to the appropriate fixed vector location.

The location of the interrupt vectors cannot be changed at runtime, nor can you change the code linked to the vector. That is, you cannot have alternate interrupt functions for the same vector and select which will be active during program execution. An error will result if there are more than one interrupt function defined for the same vector.

Interrupt vectors that have not been specified explicitly in the project can be assigned a default function address by defining an interrupt function that uses `BADISR_vect` as its vector.

### 3.9.4 Context Switching

The compiler will automatically link code into your project which saves the current status when an interrupt occurs, and restores this status when the interrupt returns.

#### 3.9.4.1 CONTEXT SAVING ON INTERRUPTS

All call-used registers will be saved in interrupt code generated by the compiler. This is the *context save* or *context switch* code.

#### 3.9.4.2 CONTEXT RESTORATION

Any objects saved by software are automatically restored by software before the interrupt function returns. The order of restoration is the reverse of that used when context is saved.

### 3.9.5 Enabling Interrupts

Two macros are available, once you have included `<xc.h>`, that control the masking of all available interrupts. These macros are `ei()`, which enable or unmask all interrupts, and `di()`, which disable or mask all interrupts.

On all devices, they affect the I bit in the status register, SREG. These macros should be used once the appropriate interrupt enable bits for the interrupts that are required in a program have been enabled.

For example:

```
TIMSK = _BV(TOIE1);
ei();       // enable all interrupts
// ...
di();       // disable all interrupts
```

**Note:** Typically you should not re-enable interrupts inside the interrupt function itself. Interrupts are automatically re-enabled by hardware on execution of the `reti` instruction. Re-enabling interrupts inside an interrupt function can result in code failure if not correctly handled.

In addition to globally enabling interrupts, each device's particular interrupt needs to be enabled separately if interrupts for this device are desired. While some devices maintain their interrupt enable bit inside the device's register set, external and timer interrupts have system-wide configuration registers.

`timer_enable_int(ints)`

This function modifies the `TIMSK` register. The value you pass via `ints` should be the bit mask for the interrupt enable bit and is device specific.

`enable_external_int(mask)`

This macro gives access to the GIMSK register (or `EIMSK` register if using an AVR Mega device or GICR register for others). This macro is unavailable if none of the registers listed above are defined.

Example:

```
// Enable timer 1 overflow interrupts
timer_enable_int(_BV(TOIE1));

// Do some work...

// Disable all timer interrupts
timer_enable_int(0);
```

### 3.9.6 Accessing Objects From Interrupt Routines

Reading or writing objects from interrupt routines can be unsafe if other functions access these same objects.

It is recommended that you explicitly mark objects accessed in interrupt and main-line code using the `volatile` specifier (see Section 3.4.8.2 “Volatile Type Qualifier”). The compiler will restrict the optimizations performed on volatile objects (see Section 3.13 “Optimizations”).

Even when objects are marked as `volatile`, the compiler cannot guarantee that they will be accessed atomically. This is particularly true of operations on multi-byte objects.

Interrupts should be disabled around any main-line code that modifies an object that is used by interrupt functions, unless you can guarantee that the access is atomic. Macros are provided in `<avr/atomic.h>` to assist you access these objects.

## 3.10 MAIN, RUNTIME STARTUP AND RESET

Coming out of reset, your program will first execute runtime startup code added by the compiler, then control is transfered to your program. This sequence is described in the following sections.

### 3.10.1 The main Function

The identifier `main` is special. You must always have one, and only one, function called `main()` in your program. This is the first C function to execute in your program.

Since your program is not called by a host, the compiler inserts special code which prevents the program crashing should the `main()` function return. This special code causes execution to jump to itself in an endless loop.

Typically your program will not terminate, and a loop construct (such as a `while(1)`) is placed around the code in your `main()` or at the end of your code, so that execution of the function will never terminate. For example:

```
int main(void)
{
    // your code goes here
    // finished that, now just wait for interrupts
    while(1)
        continue;
}
```

### 3.10.2 Runtime Startup Code

A C program requires certain objects to be initialized and the device to be in a particular state before it can begin execution of its function `main()`. It is the job of the *runtime startup* code to perform these tasks, specifically (and in no particular order):

- Initialization of static storage duration objects assigned a value when defined
- Clearing of non-initialized static storage duration objects
- General set up of registers or device state
- Calling the `main()` function

One of several runtime startup code object files which provide the runtime startup code is linked into your program.

The runtime startup code assumes that the device has just come out of Reset and that registers will be holding their power-on-reset value. Note that when the watchdog or `RST_SWRST_bm` resets the device, the registers will be reset to their known, default settings; whereas, jumping to the reset vector will not change the registers and they will be left in their previous state.

The sections used to hold the runtime startup code are listed in Table 3-8

**TABLE 3-8: RUNTIME STARTUP CODE SECTIONS USED BEFORE MAIN**

| Section name | Description |
|---|---|
| `.init0` | Weakly bound to `__init()`, see Section 3.10.3 "The Powerup Routine". If user defines `__init()`, it will be jumped into immediately after a reset. |
| `.init1` | Unused. User definable. |
| `.init2` | In C programs, weakly bound to code which initializes the stack and clears `__zero_reg__` (r1). |
| `.init3` | Unused. User definable. |
| `.init4` | This section contains the code from `libgcc.a` that copies the contents of `.data` from the program to data memory, as well as the code to clear the `.bss` section. |

**TABLE 3-8: RUNTIME STARTUP CODE SECTIONS USED BEFORE MAIN**

| Section name | Description |
|---|---|
| .init5 | Unused. User definable. |
| .init6 | Unused for C programs. |
| .init7 | Unused. User definable. |
| .init8 | Unused. User definable. |
| .init9 | Calls the main() function. |

The main() function returns to code that is also provided by the runtime startup code. You can have code executed after main() has returned by placing code in the sections listed in Table 3-9

**TABLE 3-9: RUNTIME STARTUP CODE SECTIONS USED AFTER MAIN**

| Section name | Description |
|---|---|
| .fini9 | Unused. User definable. |
| .fini8 | Unused. User definable. |
| .fini7 | In C programs, weakly bound to initialize the stack, and to clear __zero_reg__ (r1). |
| .fini6 | Unused for C program. |
| .fini5 | Unused. User definable. |
| .fini4 | Unused. User definable. |
| .fini3 | Unused for C programs. |
| .fini2 | Unused. User definable. |
| .fini1 | Unused. User definable. |
| .fini0 | Goes into an infinite loop after program termination and completion of any _exit() code (code in the .fini9 thru .fini1 sections). |

### 3.10.2.1 INITIALIZATION OF OBJECTS

One task of the runtime startup code is to ensure that any static storage duration objects contain their initial value before the program begins execution. A case in point would be input in the following example.

```
int input = 88;
```

In the above, the initial value, 0x88, will be stored as data in program memory (in the .dinit section) and will be copied to input in the data memory by the runtime startup code. For efficiency, initial values are stored as blocks of data and copied by loops. Since auto objects are dynamically created, they require code to be positioned in the function in which they are defined to perform their initialization and are not considered by the runtime startup code.

| **Note:** | Initialized auto variables can impact on code performance, particularly if the objects are large in size. Consider using static local objects instead. |
|---|---|

Objects whose content should be preserved over a Reset should be marked with the __persistent qualifier (see Section 3.4.10.8 “__persistent”). Such objects are linked in a different area of memory and are not altered by the runtime startup code.

#### 3.10.2.2 CLEARING OBJECTS

Those objects with static storage duration which are not assigned a value must be cleared before the `main()` function begins by the runtime startup code, for example.

```
int output;
```

The runtime startup code will clear all the memory locations occupied by uninitialized objects so they will contain zero before `main()` is executed.

Objects whose contents should be preserved over a Reset should be qualified with `__persistent` (see Section 3.4.9.1 "__memx Address Space Qualifier"). Such objects are linked at a different area of memory and are not altered by the runtime startup code.

### 3.10.3 The Powerup Routine

Some hardware configurations require special initialization, often within the first few instruction cycles after Reset. To achieve this you can have your own powerup routine executed during the runtime startup code.

Provided you write the required code in one of the `.initn` sections used by the runtime startup code, the compiler will take care of linking your code to the appropriate location without any need for you to adjust the linker scripts. These sections are listed in Table 3-8.

For example, the following is a small assembly sequence that is placed in the `.init1` section and is executed soon after reset and before `main()` is called.

```
#include <avr/io.h>

        .section .init1,"ax",@progbits

        ldi r16,_BV(SRE) | _BV(SRW)
        out _SFR_IO_ADDR(MCUCR),r16
```

Place this routine in an assembly source file, assemble it, and link the output with other files in your program.

Remember that code in these sections is executed before all the runtime startup code has been executed, so there is no usable stack and the `__zero_reg__` (r1) might not have been initialized. It is best to leave `__stack` at its default value (at the end of internal SRAM since this is faster and required on some devices, like the ATmega161 to work around known errata), and add `-Wl,-Tdata,0x801100` to start the data section above the stack.

## 3.11 LIBRARIES

### 3.11.1 Standard Libraries

The C standard libraries contain a standardized collection of functions, such as string, and math routines. These functions are listed in Appendix A. Library Functions.

### 3.11.2 User-Defined Libraries

User-defined libraries can be created and linked in with your program. Library files are easier to manage than many source files, and can result in faster compilation times. Libraries must, however, be compatible with the target device and options for a particular project. Several versions of a library might need to be created and maintained to allow it to be used for different projects.

Libraries can be created manually using the librarian, `xc8-ar` (see Section 4.2 “Librarian”).

Once built, user-defined libraries can be used on the command line along with the source files or added to the Libraries folder in your MPLAB X IDE project.

Library files specified on the command line are scanned first for unresolved symbols; so, their content can redefine anything that is defined in the C standard libraries (see Section 3.15.4 “Replacing Library Modules”).

### 3.11.3 Using Library Routines

Library functions and objects that have been referenced will be automatically linked into your program, provided the library file is part of your project. The use of a function from one library file will not include any other functions from that library.

Your program will require declarations for any library functions or symbols it uses. Standard libraries come with standard C headers (`.h` files), which can be included into your source files. See your favorite C text book or Appendix A. Library Functions for the header that corresponds to each library function. Typically you would write library headers if you create your own library files.

Header files are not library files. Library files contain precompiled code, typically functions and variable definitions; header files provide declarations (as opposed to definitions) for those functions, variables and types in the library. Headers can also define preprocessor macros.

## 3.12 MIXING C AND ASSEMBLY CODE

Assembly language can be mixed with C code using two different techniques: writing assembly code and placing it into a separate assembler module, or including it as in-line assembly in a C module.

> **Note:** The more assembly code a project contains, the more difficult and time consuming will be its maintenance. Assembly code might need revision if the compiler is updated due to differences in the way the updated compiler may work. These factors are less likely to occur if the code is written in C. If assembly must be added, it is preferable to write this as a self-contained routine in a separate assembly module, rather than in-lining it in C code.

### 3.12.1 Integrating Assembly Language Modules

Entire functions can be coded in assembly language as separate `.s` (or `.S` or `.sx`) source files included into your project. They will be assembled and combined into the output image by the linker.

The following are guidelines that must be adhered to when writing a C-callable assembly routine.

- Include the `<xc.h>` header file in your code. If this is included using `#include`, ensure the extension used by the source file is `.S` or `.sx` to ensure the file is pre-processed.
- Select or define a suitable section for the executable assembly code (see Section 3.15.1 "Compiler-Generated Psects" for an introductory guide).
- Select a name (label) for the routine
- Ensure that the routine's label is accessible from other modules
- Use macros like `_SFR_IO_ADDR` to obtain the correct SFR address to use with instructions that can access the IO memory space.
- Select an appropriate C-equivalent prototype for the routine on which argument passing can be modeled.
- If values need to be passed to or returned from the routine, use the appropriate registers to passed the arguments.

The following example shows an assembly routine for an atmega103 device that takes an `int` parameter, adds this to the content of PORTD, and returns this as an `int`.

```
#include <xc.h>

        .section .text
        .global plus     ; allow routine to be externally used
plus:
; int parameter in r24/5
        in      r18, _SFR_IO_ADDR(PORTD)        ; read PORTD
        add     r24, r18        ; add to parameter
        adc     r25, r1         ; add zero to MSB
; parameter registers are also the return location, so ready to return
        ret
        .end
```

The code has been placed in a `.text` section, so it will be automatically placed in the area of memory set aside for code without you having to adjust the default linker options.

The `_SFR_IO_ADDR` macro has been used to ensure that the correct address was specified to instructions that read the IO memory space.

Because the C preprocessor `#include` directive and preprocessor macros were used, the assembly file must be preprocessed to ensure it uses a `.S` or `.sx` extension when compiled.

To call an assembly routine from C code, a declaration for the routine must be provided. Here is a C code snippet that declares the operation of the assembler routine, then calls the routine.

```
// declare the assembly routine so it can be correctly called
extern int plus(int);

void main(void) {
  volatile unsigned int result;

  result = plus(0x55);  // call the assembly routine
}
```

### 3.12.2 In-line Assembly

Assembly instructions can be directly embedded in-line into C code using the statement `asm();`. In-line assembly has two forms: simple and extended.

In the simple form, the assembler instruction is written using the syntax:

`asm("`*`instruction`*`");`

where *`instruction`* is a valid assembly-language construct, for example:

```
asm("sei");
```

You can write several instructions in the one string, but you should put each instruction on a new line and use linefeed and tab characters to ensure they are properly formatted in the assembly listing file.

```
asm ("nop\n\t"
     "nop\n\t"
     "nop\n\t"
     "nop\n\t");
```

In an extended assembler instruction using `asm()`, the operands of the instruction are specified using C expressions. The extended syntax, discussed in the following sections, has the general form:

```
asm("template" [ : [ "constraint"(output-operand) [ , ... ] ]
               [ : [ "constraint"(input-operand) [ , ... ] ]
               [ "clobber" [ , ... ] ]
               ] ]);
```

For example,

```
asm("in %0, %1" : "=r" (value) : "I" (_SFR_IO_ADDR(PORTD)) );
```

The template specifies the instruction mnemonic and optional placeholders for the input and output operands, specified by a percent sign followed by a single digit and which are described in the following section. The compiler replaces these and other tokens in the template that refer to inputs, outputs, and goto labels, then outputs the resulting string to the assembler.

#### 3.12.2.1 INPUT AND OUTPUT OPERANDS

Following the template is a comma-separated list of zero or more output operands, which indicate the names of C objects modified by the assembly code and input operands, which make values from C variables and expressions available to the assembly code.

Each operands has several components, described by:

`[ [asmSymbolicName] ] constraint (Cexpression)`

where *asmSymbolicName* is an optional symbolic name for the operand, *constraint* is string specifying constraints on the placement of the operand, and *Cexpression* is the C variable or expression to be used by the operand and which is enclosed in parentheses.

The first (left-most) output operand is numbered 0, any subsequent output operands are numbered one higher than the operand before it, with input operands being numbered in the same way.

The supported constraint letters are listed in Table 3-10; modifiers in Table 3-12.

**TABLE 3-10: INPUT AND OUTPUT OPERAND CONSTRAINTS**

| Letter | Constraint | Range |
|---|---|---|
| a | Simple upper registers | r16 to r23 |
| b | Base pointer registers pairs | r28 to r32 (Y, Z) |
| d | Upper register | r16 to r31 |
| e | Pointer register pairs | r26 to r31 (X, Y, Z) |
| l | Lower registers | r0 to r15 |
| q | Stack pointer register | SPH:SPL |
| r | Any register | r0 to r31 |
| t | Temporary register | r0 |
| w | Special upper register pairs usable in `adiw` instruction | r24, r26, r28, r30 |
| x | Pointer register pair X | r27:r26 (X) |
| y | Pointer register pair Y | r29:r28 (Y) |
| z | Pointer register pair Z | r31:r30 (Z) |
| G | Floating point constant | 0.0 |
| I | 6-bit positive integer constant | 0 to 63 |
| J | 6-bit negative integer constant | -63 to 0 |
| K | Integer constant | 2 |
| L | Integer constant | 0 |
| M | 8-bit integer constant | 0 to 255 |
| N | Integer constant | -1 |
| O | Integer constant | 8, 16, 24 |
| P | Integer constant | 1 |
| Q | Memory address based on Y or Z pointer with displacement | |
| Cm2 | Integer constant | -2 |
| C0*n* | Integer constant, where *n* ranges from 0 to 7 | *n* |
| Ca*n* | *n*-byte integer constant that allows AND without clobber register, where *n* ranges from 2 to 4 | |
| Co*n* | *n*-byte integer constant that allows OR without clobber register, where *n* ranges from 2 to 4 | |
| Cx*n* | *n*-byte integer constant that allows XOR without clobber register, where *n* ranges from 2 to 4 | |
| Csp | Integer constant | -6 to 6 |
| Cxf | 4-byte integer constant with at least one 0xF nibble | |
| C0f | 4-byte integer constant with no 0xF nibbles | |

**TABLE 3-10: INPUT AND OUTPUT OPERAND CONSTRAINTS**

| Letter | Constraint | Range |
|---|---|---|
| Ynn | Fixed-point constant known at compile time | |
| Y0*n* | Fixed-point or integer constant, where *n* ranges from 0 to 2 | *n* |
| Ym*n* | Fixed-point or integer constant, where *n* ranges from 1 to 2 | -*n* |
| YIJ | Fixed-point or integer constant | -0x3F to 0x3F |

The constraint you choose should match the registers or constants that are appropriate for the AVR instruction operand. The compiler will check the constraint against your C expression; however, if the wrong constraint is used, there is the possibility of code failing at runtime. For example, if you specify the constraint `r` with an `ORI` instruction, then the compiler is free to select any register (r0 thru r31) for that operand. This will fail, if the compiler chooses a register in the range r2 to r15. The correct constraint in this case is `d`. On the other hand, if you use the constraint `M`, the compiler will make sure that you only use an 8-bit immediate value operand.

Table 3-11 shows all the AVR assembler mnemonics that require operands and the relevant constraints (explained in Table 3-10) for each of those operands.

**TABLE 3-11: INSTRUCTIONS AND OPERAND CONSTRAINTS**

| Mnemonic | Constraints | Mnemonic | Constraints |
|---|---|---|---|
| adc | r,r | add | r,r |
| adiw | w,I | and | r,r |
| andi | d,M | asr | r |
| bclr | I | bld | r,I |
| brbc | I,label | brbs | I,label |
| bset | r,I | bst | r,I |
| cbi | I,I | cbr | d,I |
| com | r | cp | r,r |
| cpc | r,r | cpi | d,M |
| cpse | r,r | dec | r |
| elpm | t,z | eor | r,r |
| in | r,I | inc | r |
| ld | r,e | ldd | r,b |
| ldi | d,M | lds | r,label |
| lpm | t,z | lsl | r |
| lsr | r | mov | r,r |
| movw | r,r | mul | r,r |
| neg | r | or | r,r |
| ori | d,M | out | I,r |
| pop | r | push | r |
| rol | r | ror | r |
| sbc | r,r | sbci | d,M |
| sbi | I,I | sbic | I,I |
| sbiw | w,I | sbr | d,M |
| sbrc | r,I | sbrs | r,I |
| ser | d | st | e,r |
| std | b,r | sts | label,r |

**TABLE 3-11: INSTRUCTIONS AND OPERAND CONSTRAINTS**

| Mnemonic | Constraints | Mnemonic | Constraints |
|---|---|---|---|
| sub | r,r | subi | d,M |
| swap | r | | |

Constraint characters may be prepended by a single constraint modifier. Constraints without a modifier specify read-only operands. The constraint modifiers are shown in Table 3-12

**TABLE 3-12: INPUT AND OUTPUT CONSTRAINT MODIFIERS**

| Letter | Constraint |
|---|---|
| = | Write-only operand, usually used for all output operands. |
| + | Read-write operand |
| & | Register should be used for output only |

So, in the example:

```
asm("in %0, %1" : "=r" (value) : "I" (_SFR_IO_ADDR(PORTD)) );
```

the assembler instruction is defined by the template, `"in %0, %1"`. The `%0` token refers to the first output operand, `"=r" (value)`, and `%1` refers to the first input operand, `"I" (_SFR_IO_ADDR(PORTD))`. No clobbered registers were indicated in this example.

The compiler might encode the above in-line assembly as follows:

```
lds r24,value
/* #APP */
in r24, 12
/* #NOAPP */
sts value,r24
```

The comments have been added by the compiler to inform the assembler that the enclosed instruction was hand-written. In this example, the compiler selected register r24 for storage of the value read from `PORTD`; however, it might not explicitly load or store the value, nor include your assembler code at all, based on the compiler's optimization strategy. For example, if you never use the variable `value` in the remaining part of the C program, the compiler could remove your in-line assembly code unless you switch off the optimizers. To avoid this, you can add the `volatile` attribute to the asm statement, as shown below:

```
asm volatile("in %0, %1" : "=r" (value) : "I" (_SFR_IO_ADDR(PORTD)));
```

Operands can be given names, if desired. The name is prepended in brackets to the constraints in the operand list and references to the named operand use the bracketed name instead of a number after the `%` sign. Thus, the above example could also be written as

```
asm("in %[retval], %[port]" :
    [retval] "=r" (value) :
    [port] "I" (_SFR_IO_ADDR(PORTD)) );
```

The clobber list is primarily used to tell the compiler about modifications done by the assembler code. This section of the statement can be omitted, but all other sections are required. Use the delimiting colons, but leave the operand field empty if there is no input or output used, for example:

```
asm volatile("cli"::);
```

Output operands must be write-only and the C expression result must be an lvalue, i.e., be valid on the left side of an assignment. Note, that the compiler will not check if the operands are of a reasonable type for the kind of operation used in the assembler instructions. Input operands are read-only.

In cases where you need the same operand for input and output, read-write operands are not supported, but it is possible to indicate which operand's register to use as the input register by a single digit in the constraint string. Here is an example:

```
asm volatile("swap %0" : "=r" (value) : "0" (value));
```

This statement will swap the nibbles of an 8-bit variable named `value`. Constraint `"0"` tells the compiler, to use the same input register used by the first operand. Note, however, that this doesn't automatically imply the reverse case.

The compiler may choose the same registers for input and output, even if not told to do so. This can be an issue if the output operand is modified by the assembler code before the input operand is used. In the situation where your code depends on different registers used for input and output operands, you must use the constraint modifier, `&`, with the output operand, as shown in the following example.

```
asm volatile("in %0,%1"    "\n\t"
             "out %1,%2"   "\n\t"
             : "=&r" (result)
             : "I" (_SFR_IO_ADDR(port)), "r" (source)
            );
```

Here, a value is read from a port and then a value is written to the same port. If the compiler choses the same register for input and output, then the output value will be clobbered by the first assembler instruction; however, the use of the `&` constraint modifier prevents the compiler from selecting any register for the output value that is also used for any of the input operands.

Here is another example that swaps the high and low byte of a 16-bit value:

```
asm volatile("mov __tmp_reg__, %A0" "\n\t"
             "mov %A0, %B0"         "\n\t"
             "mov %B0, __tmp_reg__" "\n\t"
             : "=r" (value)
             : "0" (value)
            );
```

Notice the usage of register `__tmp_reg__`, which you can use without having to save its content. The letters `A` and `B`, used in the tokens representing the instruction operands refer to byte components of a multi-byte register, `A` referring to the least significant byte, `B` the next most significant byte, etc.

The following example, which swaps bytes of a 32-bit value, uses the C and D components of a 4 byte quantity, and rather than list the same operand as both input and output operand (via `"0"` as the input operand constraint), it can also be declared as a read-write operand by using `"+r"` as the output constraint.

```
asm volatile("mov __tmp_reg__, %A0" "\n\t"
             "mov %A0, %D0"         "\n\t"
             "mov %D0, __tmp_reg__" "\n\t"
             "mov __tmp_reg__, %B0" "\n\t"
             "mov %B0, %C0"         "\n\t"
             "mov %C0, __tmp_reg__" "\n\t"
             : "+r" (value)
            );
```

If operands do not fit into a single register, the compiler will automatically assign enough registers to hold the entire operand. This also implies, that it is often necessary to cast the type of an input operand to the desired size.

If an input operand constraint indicates a pointer register pair, such as `"e" (ptr)`, and the compiler selects register Z (r30:r31), then you must use `%a0` (lower case `a`) to refer to the Z register, when used in a context like:

```
ld r24,Z
```

#### 3.12.2.2 CLOBBER OPERAND

The list of clobbered registers is optional; however, if the instruction modifies registers that are not specified as operands, you need to inform the compiler of these changes.

Typically you can arrange the assembly so that you do not need to specify what has been clobbered. Indicating that a register has been clobbered will force the compiler to store their values before and reload them after your assembly instructions and will limit the ability of the compiler to optimize your code.

The following example will perform an atomic increment. It disables the interrupts then increments an 8-bit value pointed to by a pointer variable. Note, that a pointer is used because the incremented value needs to be stored before the interrupts are enabled.

```
asm volatile(
    "cli"               "\n\t"
    "ld r24, %a0"       "\n\t"
    "inc r24"           "\n\t"
    "st %a0, r24"       "\n\t"
    "sei"               "\n\t"
    :
    : "e" (ptr)
    : "r24"
);
```

The compiler might produce the following code for the above:

```
cli
ld r24, Z
inc r24
st Z, r24
sei
```

To have this sequence avoid clobbering register r24, make use of the special temporary register `__tmp_reg__` defined by the compiler.

```
asm volatile(
    "cli"                       "\n\t"
    "ld __tmp_reg__, %a0"       "\n\t"
    "inc __tmp_reg__"           "\n\t"
    "st %a0, __tmp_reg__"       "\n\t"
    "sei"                       "\n\t"
    :
    : "e" (ptr)
);
```

The compiler will always reload the temporary register when it is needed.

The above code unconditionally re-enables the interrupts, which may not be desirable. To make the code more versatile, the current status can be stored in a register selected by the compiler.

```
{
    uint8_t s;
    asm volatile(
        "in %0, __SREG__"           "\n\t"
        "cli"                       "\n\t"
        "ld __tmp_reg__, %a1"       "\n\t"
        "inc __tmp_reg__"           "\n\t"
        "st %a1, __tmp_reg__"       "\n\t"
        "out __SREG__, %0"          "\n\t"
        : "=&r" (s)
        : "e" (ptr)
    );
}
```

The assembler code here modifies the variable, that `ptr` points to, so the definition of `ptr` should indicate that its target can change unexpected, using the `volatile` specifier, for example:

```
volatile uint8_t *ptr;
```

The special clobber `memory` informs the compiler that the assembler code may modify any memory location. It forces the compiler to update all variables for which the contents are currently held in a register before executing the assembler code.

When you use a memory clobber with an assembly instruction, it ensures that all prior accesses to volatile objects are complete before the instruction executes, and that execution of volatile accesses after the instruction have not commenced. However, it does not prevent the compiler from moving non-volatile-related instructions across the barrier created by the memory clobber instruction, as such instructions might be those that enable or disable interrupts.

#### 3.12.2.3 ASSEMBLY MACROS

In-line assembly language code sequences can be defined as macros and provided in header files so that they may be reused. Some are already defined in the library file headers, found in the `avr/avr/include` directory.

Using include files that contain assembly macros might produce compiler warnings when compiling in strict ANSI mode. To avoid that, you can use the `__asm__` and `__volatile__` keyword aliases.

If a macro contains the definition for a label and that macro is used several times, this can result in multiply defined symbols. In such cases, you can use a special pattern, `%=`, which is replaced by a unique number on each asm statement, as shown in the following example:

```
#define loop_until_bit_is_clear(port,bit) \ __asm__ __volatile__ ( \
"L_%=: " "sbic %0, %1" "\n\t" \ "rjmp L_%=" \ : /* no outputs */ : "I"
(_SFR_IO_ADDR(port)), "I" (bit) )
```

When used for the first time, `L_=` might be replaced by `L_1404`, the next usage might generate `L_1405` or whatever.

Another option is to use Unix-assembler style numeric labels, which consist of a number only. References to these labels consist of the number, followed by the letter `b` for a backward reference, or `f` for a forward reference. These local labels may be re-used within the source file, references will pick the closest label with the same number and given direction. Using these labels, the above example becomes:

```
#define loop_until_bit_is_clear(port,bit) __asm__ __volatile__ ( "1: "
"sbic %0, %1" "\n\t" "rjmp 1b" : /* no outputs */ : "I"
(_SFR_IO_ADDR(port)), "I" (bit) )
```

### 3.12.3 Interaction between Assembly and C Code

MPLAB XC8 C Compiler incorporates several features designed to allow C code to obey requirements of user-defined assembly code. There are also precautions that must be followed to ensure that assembly code does not interfere with the assembly generated from C code.

#### 3.12.3.1 EQUIVALENT ASSEMBLY SYMBOLS

By default AVR-GCC uses the same symbolic names of functions or objects in C and assembler code. There is no leading underscore character prepended to a C language's symbol in assembly code.

You can specify a different name for the assembler code by using a special form of the `asm()` statement:

```
unsigned long value asm("clock") = 3686400;
```

This statement instructs the compiler to use the symbol name `clock` rather than `value`. This makes sense only for objects with static storage duration, because stack-based objects do not have symbolic names in the assembler code and these can be cached in registers.

With the compiler you can specify the use of a specific register:

```
void Count(void)
{
    register unsigned char counter asm("r3");

    // ... some code...
    asm volatile("clr r3");
    // ... more code...
}
```

The assembler instruction, `clr r3`, will clear the variable `counter`. The compiler will not completely reserve the specified register, and it might be re-used. The compiler is unable to check whether the use of the specified register conflicts with any other predefined register. It is recommended that you do not reserve too many registers in this way.

In order to change the assembly name of a function, you need a prototype declaration, because the compiler will not accept the `asm()` keyword in a function definition. For example:

```
extern long calc(void) asm ("CALCULATE");
```

Calling the function `calc()` in C code will generate assembler instructions which call the function `CALCULATE()`.

#### 3.12.3.2 ACCESSING REGISTERS FROM ASSEMBLY CODE

In assembly code, SFR definitions are not automatically accessible. The header file `<xc.h>` can be included to gain access to these register definitions.

The symbols for registers in this header file are the same as those used in the C domain; however, you should use the appropriate I/O macros to ensure the correct address is encoded into instructions which accesses memory in the I/O space, for example to following writes to the TCNT0 register:

```
out     _SFR_IO_ADDR(TCNT0), r19
```

Bits within registers have macros associated with them and can be used directly with instructions that expect a bit number (0 thru 7), or with the `_BV()` macro if you need a bit mask based on that bit's position in the SFR, for example:

```
sbic    _SFR_IO_ADDR(PORTD), PD4
ldi     r16, _BV(TOIE0)
```

## 3.13 OPTIMIZATIONS

The MPLAB XC8 compiler can perform a variety of optimizations. Optimizations can be controlled using the -O option (described in Section 2.7.6 “Options for Controlling Optimization”). In Free mode, some of these optimizations are disabled. Even if they are enabled, optimizations might only be applied if very specific conditions are met. As a result, you might see that some lines of code are optimized, but other similar lines are not. When debugging code, you may wish to reduce the optimization level to ensure expected program flow.

## 3.14 PREPROCESSING

All C source files are preprocessed before compilation. The `-E` option can be used to preprocess and then stop the compilation (see Section 2.7.2.2 “E: Preprocess Only”).

Assembler files can also be preprocessed if they use a `.S` or `.sx` extension.(see Section 2.2.3 “Input File Types”).

### 3.14.1 Preprocessor Directives

MPLAB XC8 accepts the standard preprocessor directives, and these are listed in Table 3-13.

Macro expansion using arguments can use the `#` character to convert an argument to a string, and the `##` sequence to concatenate arguments. If two expressions are being concatenated, consider using two macros in case either expression requires substitution itself; for example

```
#define __paste1(a,b)   a##b
#define __paste(a,b)    __paste1(a,b)
```

You can use the `paste` macro to concatenate two expressions that themselves can require further expansion. Once a macro identifier has been expanded, it will not be expanded again if it appears after concatenation.

**TABLE 3-13: PREPROCESSOR DIRECTIVES**

| Directive | Meaning | Example |
|---|---|---|
| # | preprocessor null directive, do nothing | # |
| #assert | generate error if condition false | #assert SIZE > 10 |
| #define | define preprocessor macro | #define SIZE (5)<br>#define FLAG<br>#define add(a,b) ((a)+(b)) |
| #elif | short for #else #if | see #ifdef |
| #else | conditionally include source lines | see #if |
| #endif | terminate conditional source inclusion | see #if |
| #error | generate an error message | #error Size too big |
| #if | include source lines if constant expression true | #if SIZE < 10<br>  c = process(10)<br>#else<br>  skip();<br>#endif |
| #ifdef | include source lines if preprocessor symbol defined | #ifdef FLAG<br>  do_loop();<br>#elif SIZE == 5<br>  skip_loop();<br>#endif |
| #ifndef | include source lines if preprocessor symbol not defined | #ifndef FLAG<br>  jump();<br>#endif |
| #include | include text file into source | #include <stdio.h><br>#include "project.h" |
| #line | specify line number and filename for listing | #line 3 final |
| #*nn* | (where *nn* is a number) short for #line *nn* | #20 |
| #pragma | compiler specific options | Refer to Section 3.14.3 "Pragma Directives" |
| #undef | undefines preprocessor symbol | #undef FLAG |
| #warning | generate a warning message | #warning Length not set |

#### 3.14.1.1 PREPROCESSOR ARITHMETIC

Preprocessor macro replacement expressions are textual and do not utilize types. Unless they are part of the controlling expression to the inclusion directives (discussed below), macros are not evaluated by the preprocessor. Once macros have been textually expanded and preprocessing is complete, the expansion forms a C expression which is evaluated by the *code generator* along with other C code. Tokens within the expanded C expression inherit a type and values are then subject to integral promotion and type conversion in the usual way.

If a macro is part of the controlling expression to a conditional inclusion directive (#if or #elif), the macro must be evaluated by the *preprocessor*. The result of this evaluation is often different to the C-domain result for the same sequence. The preprocessor assigns sizes to literal values in the controlling expression that are equal to the largest integer size accepted by the compiler. For the MPLAB XC8 C compiler, this size is 64 bits.

The following code does not work as you might expect it to work. The preprocessor will evaluate `MAX` to be the result of a 64-bit multiplication, 0xF4240. However, the definition of the `long int` variable, `max`, will be assigned the value 0x4240 (since the constant 1000 has a `signed int` type, and the C-domain multiplication will also be performed using a 16-bit `signed int` type).

```
#define MAX 1000*1000
...
#if MAX > INT16_MAX   // evaluation of MAX by preprocessor
long int max = MAX;   // evaluation of MAX by code generator
#else
int max = MAX;        // evaluation of MAX by code generator
#endif
```

Overflow in the C domain can be avoided by using a constant suffix in the macro (see Section 3.4.7 “Constant Types and Formats”). For example, an `L` after a number in a macro expansion indicates it should be interpreted by the C compiler as a `long`, but this suffix does not affect how the preprocessor interprets the value, if it needs to evaluate it.

So, for example:

```
#define MAX 1000*1000L
```

will evaluate to 0xF4240 in C expressions.

### 3.14.2 Predefined Macros

The compiler drivers define certain symbols to the preprocessor, allowing conditional compilation based on chip type, etc. The symbols listed in Table 3-14 show the more common symbols defined by the drivers. Each symbol, if defined, is equated to 1. (unless otherwise stated).

**TABLE 3-14: PREDEFINED MACROS**

| Symbol | Description |
|---|---|
| _XC8_MODE_ | indicates which compiler, PRO, Standard or Free, is in use Values of 2, 1 or 0 are assigned, respectively. |
| __AVR_*Device*__ | Set when the -mcpu option specifies a device rather than an architecture. It indicates the device, for example when compiling for an atmega8, the macro __AVR_ATmega8__ will be set. |
| __AVR_DEVICE_NAME__ | Set when the -mcpu option specifies a device rather than an architecture. It indicates the device, for example when compiling for an atmega8 the macro is defined to atmega8. |
| __AVR_ARCH__ | Indicates the device architecture. Possible values are: 2, 25, 3, 31, 35, 4, 5, 51, 6 for the avr2, avr25, avr3, avr31, avr35, avr4, avr5, avr51, avr6, architectures respectively and 100, 102, 103, 104, 105, 106, 107 for the avrtiny, avrxmega2, avrxmega3, avrxmega4, avrxmega5, avrxmega6, avrxmega7, architectures respectively. |
| __AVR_ASM_ONLY__ | Indicates that the selected device can only be programmed in assembly. |
| __AVR_ERRATA_SKIP__ __AVR_ERRATA_SKIP_JMP_CALL__ | Indicates the selected device (AT90S8515, ATmega103) must not skip (SBRS, SBRC, SBIS, SBIC, and CPSE instructions) 32-bit instructions because of a hardware erratum. The second macro is only defined if __AVR_HAVE_JMP_CALL__ is also set. |
| __AVR_HAVE_EIJMP_EICALL__ | Indicates the selected device has more then 128 kB of program memory, a 3-byte wide program counter, and the EIJMP and EICALL instructions. |
| __AVR_HAVE_ELPM__ | Indicates the selected device has the ELPM instruction. |
| __AVR_HAVE_ELPMX__ | Indicates the device has the ELPM Rn,Z and ELPM Rn,Z+ instructions. |
| __AVR_HAVE_JMP_CALL__ | Indicates the selected device has the JMP and CALL instructions and has more than 8kB of program memory. |
| __AVR_HAVE_LPMX__ | Indicates the selected device has the LPM Rn,Z and LPM Rn,Z+ instructions. |
| __AVR_HAVE_MOVW__ | Indicates the selected device has the MOVW instruction, to perform 16-bit register-register moves. |
| __AVR_HAVE_MUL__ __AVR_HAVE_MUL__ | Indicates the selected device has a hardware multiplier. |
| __AVR_HAVE_RAMPD__ __AVR_HAVE_RAMPX__ __AVR_HAVE_RAMPY__ __AVR_HAVE_RAMPZ__ | Indicates the device has the RAMPD, RAMPX, RAMPY, or RAMPZ special function register, respectively. |
| __AVR_HAVE_SPH__ __AVR_SP8__ | Indicates the device has a 16- or 8-bit stack pointer, respectively. The definition of these macros is affected by the selected device, and for avr2 and avr25 architectures. |

**TABLE 3-14: PREDEFINED MACROS**

| Symbol | Description |
|---|---|
| __AVR_HAVE_8BIT_SP__ __AVR_HAVE_16BIT_SP__ | Indicates the whether 8- or 16-bits of the stack pointer is used, respectively, by the compiler. The `-mtiny-stack` option will affect which macros are defined |
| __AVR_ISA_RMW__ | Indicates the selected device has Read-Modify-Write instructions (XCH, LAC, LAS and LAT). |
| __AVR_MEGA__ | Indicates the selected devices JMP and CALL instructions. |
| __AVR_PM_BASE_ADDRESS__=addr | Indicates the address space is linear and program memory is mapped into data memory. The value assigned to this macro is the starting address of the mapped memory. |
| __AVR_SFR_OFFSET__=offset | Indicates the offset to subtract from the data memory address for those instructions (e.g. IN, OUT, and SBI) that can access SFRs directly. |
| __AVR_SHORT_CALLS__ | Indicates the use of the `-mshort-calls` option, which affects the call instruction used and which can be set automatically. |
| __AVR_TINY__ | Indicates that the selected device or architecture belongs to the TINY family. |
| __AVR_TINY_PM_BASE_ADDRESS__=addr | Deprecated; use `__AVR_PM_BASE_ADDRESS__`. Indicates the TINY device address space is linear and program memory is mapped into data memory. |
| __AVR_XMEGA__ | Indicates that the selected device or architecture belongs to the XMEGA family. |
| __AVR_2_BYTE_PC__ | Indicates the selected device has up to 128 kB of program memory and the program counter is 2 bytes wide. |
| __AVR_3_BYTE_PC__ | Indicates the selected device has at least 128 kB of program memory and the program counter is 3 bytes wide. |
| __BUILTIN_AVR_*name* | Indicates the names built-in feature is available for the selected device |
| __FLASH*n* | Defines `__FLASH`, `__FLASH1`, `__FLASH2` etc, based on the number of flash segments on the selected device. |
| __MEMX | Indicates the `__memx` specifier is available for the selected device. |
| __NO_INTERRUPTS__ | Indicates the use of the `-mno-interrupts` option, which affects how the stack pointer is changed. |
| __DATE__ | Indicate the current date, e.g., `May 21 2004` |
| __FILE__ | Indicate this source file being preprocessed |
| __TIME__ | Indicate the current time, e.g., `08:06:31` |
| __XC | Indicates MPLAB XC compiler for Microchip is in use |
| __XC8 | Indicates MPLAB XC compiler for Microchip 8-bit devices is in use |
| __XC8_VERSION | Indicates the compiler's version number multiplied by 1000, e.g., v1.00 will be represented by 1000 |

### 3.14.3 Pragma Directives

There is only one MPLAB XC8-specific pragma that is accepted, the `config` pragma (discussed in Section 3.3.5 "Configuration Bit Access").

## 3.15 LINKING PROGRAMS

The compiler will automatically invoke the linker unless the compiler has been requested to stop earlier in the compilation sequence.

The linker will run with options that are obtained from the command-line driver and use linker scripts, which specify memory areas and where sections are to be placed.

The linker can create a map file which details the memory assigned to sections and objects. The map file is the best place to look for memory information.

### 3.15.1 Compiler-Generated Psects

The code generator places code and data into sections with standard names, which are subsequently positioned by the default linker scripts. A section can be created in assembly code by using the `.section` assembler directive. If you are unsure which section holds an object or code in your project, produce and check an assembly list file.

The contents of common sections are described below.

#### 3.15.1.1 PROGRAM SPACE SECTIONS

`.text` – These sections contain all executable code that does not require a special link location.

`.init`*n* These sections are used to define the runtime startup code, executed from the moment of reset right through to the invocation of `main()`. The code in these sections are executed in order from `init0` to `init9`.

`.fini`*n* These sections are used to define the exit code, executed after `main()` terminates, either by returning or by calling to `exit()`. The code in the `.finiN` sections are executed in descending order from `.fini9` to `.fini0`.

#### 3.15.1.2 DATA SPACE SECTIONS

`.bss` – This section contains any objects with static storage duration that have not been initialized.

`.data` – This section contains the RAM image of any objects with static storage duration that have been initialized with values.

`.rodata` These sections hold read-only data.

### 3.15.2 Changing and Linking the Allocated Section

The location of the default sections in which functions and objects are placed can be changed via driver options. Section 3.15.1 “Compiler-Generated Psects” lists the default sections the compiler uses to hold objects and code.

The `__section()` specifier allows you to have a object or function redirected into a user-define section. This allows you to relocate individual objects or functions.

Objects that use the `__section()` specifier will be cleared or initialized in the usual way by the runtime startup code.

The following are examples of a object and function allocated to a non-default section.

```
int __section("myBss") foobar;
int __section("myText") helper(int mode) { /* ... */ }
```

You can link these sections by using the `-Wl,--section-start=`*`section=addr`* option when building (linking) your program, provided that the linker script has already defined an output section with the same name. Note that you need to use an offset of 0x800000 for any address that is in the data space. For example, suppose you wish to place the new `myBss` section, created above, at SRAM address 0x300:

```
-Wl,--section-start=myBss=0x800300
```

For standard sections, like the `.text`, `.data` and `.bss` sections, they can be positioned using the `-Wl,-T`*`section,addr`* option when building (linking) your program. Thus, if you want the `.data` section to start at 0x1100, you can use the following option:

```
-Wl,-Tdata=0x801100
```

### 3.15.3 Linker Scripts

Linker scripts are used to instruct the linker how to position sections in memory. There are five different variants of these scripts, shown in Table 3-15, which are selected based on the options passed to the linker.

**TABLE 3-15: LINKER SCRIPT VARIANTS**

| Script Extension | Controlling linker option | Linker operation |
|---|---|---|
| `.x` | default | |
| `.xr` | `-r` | perform no relocation |
| `.xu` | `-Ur` | resolve references to constructors |
| `.xn` | `-n` | set text to be read-only |
| `.xbn` | `-N` | Set the text and data sections to be readable and writable. |

### 3.15.4 Replacing Library Modules

For library functions that are weak (see Section 3.8.2.3 “weak”), you can have your own version of a routine replace a library routine with the same name without having to using the librarian, `,xc8-ar` (see Section 4.2 “Librarian”). Simply include the definition of that routine as part of your project.

**NOTES:**

# Chapter 4. Utilities

## 4.1 INTRODUCTION

This chapter discusses some of the utility applications that are bundled with the compiler.

The applications discussed in this chapter are those more commonly used, but you do not typically need to execute them directly. Most of their features are invoked indirectly by the command line driver that is based on the command-line arguments or MPLAB X IDE project property selections.

The following applications are described in this chapter of the MPLAB XC8 C Compiler User's Guide:

- Librarian
- Hexmate
- Objdump

## 4.2 LIBRARIAN

The librarian program, `xc8-ar`, has the function of combining several intermediate files into a single file, known as a library or archive file. Libraries are easier to manage and might consume less disk space that the individual files.

The librarian can build all library types needed by the compiler and can detect the format of existing libraries.

### 4.2.1 Using the Librarian

The librarian program is called `xc8-ar` and has the following basic command format:

```
xc8-ar [options] file.a [file1.o file2.o...]
```

where *`file.a`* represents the library being created or edited. The following files, if required, are the modules of the library that is required by the command specified.

The *`options`* is zero or more options, shown in Table 4-1, that control the program.

**TABLE 4-1: LIBRARIAN COMMAND-LINE OPTIONS**

| Option | Effect |
|---|---|
| `-d` | Delete module |
| `-m` | Re-order modules |
| `-p` | List modules |
| `-r` | Replace modules |
| `-x` | Extract modules |
| `--target` | Specify output format |

When replacing or extracting modules, the names of the modules to be replaced or extracted must be specified. If no names are supplied, all the modules in the library will be replaced or extracted respectively.

Creating a library file or adding a file to an existing library is performed by requesting the librarian to replace the module in the library. Since the module is not present, it will be appended to the library. The librarian creates libraries with the modules in the order in which they were given on the command line. When updating a library, the order of the modules is preserved. Any modules added to a library will be appended to the end.

The ordering of the modules in a library is significant to the linker. If a library contains a module that references a symbol defined in another module in the same library, the module defining the symbol should come after the module referencing the symbol.

When using the `-d` option, the specified modules will be deleted from the library. In this instance, it is an error not to supply any module names.

The `-p` option will list the modules within the library file.

The `-m` option takes a list of module names and re-orders the matching modules in the library file so that they have the same order as the one listed on the command line. Modules that are not listed are left in their existing order and will appear after the re-ordered modules.

#### 4.2.1.1 EXAMPLES

Here are some examples of usage of the librarian. The following command:

```
xc8-ar -r myPicLib.a ctime.o init.o
```

creates a library called `myPicLib.a` that contains the modules `ctime.o` and `init.o`

The following command deletes the object module `a.o` from the library `lcd.a`:

```
xc8-ar -d lcd.a a.o
```

## 4.3 HEXMATE

The `hexmate` utility is a program designed to manipulate Intel HEX files. `hexmate` is a post-link stage utility that provides the facility to:

- Calculate and store variable-length hash values
- Fill unused memory locations with known data sequences
- Merge multiple Intel HEX files into one output file
- Convert INHX32 files to other INHX formats (e.g., INHX8M)
- Detect specific or partial opcode sequences within a HEX file
- Find/replace specific or partial opcode sequences
- Provide a map of addresses used in a HEX file
- Change or fix the length of data records in a HEX file
- Validate checksums within Intel HEX files

Typical applications for `hexmate` might include:

- Merging a bootloader or debug module into a main application at build time
- Calculating a checksum or CRC value over a range of program memory and storing its value in program memory or EEPROM
- Filling unused memory locations with an instruction to send the PC to a known location if it gets lost
- Storage of a serial number at a fixed address
- Storage of a string (e.g., time stamp) at a fixed address
- Store initial values at a particular memory address (e.g., initialize EEPROM)
- Detecting usage of a buggy/restricted instruction
- Adjusting HEX file to meet requirements of particular bootloaders

### 4.3.1 Hexmate Command Line Options

If `hexmate` is to be run directly, its usage is:

```
hexmate [specs,]file1.hex [... [specs,]fileN.hex] [options]
```

where *`file1.hex`* through to *`fileN.hex`* form a list of input Intel HEX files to merge.

If only one HEX file is specified, no merging takes place, but other functionality can be specified by additional options. Table 4-2 lists the command line options that `hexmate` accepts.

**TABLE 4-2: `hexmate` COMMAND-LINE OPTIONS**

| Option | Effect |
|---|---|
| `--EDF` | Specify the message description file. |
| `--EMAX` | Set the maximum number of permitted errors before terminating. |
| `--MSGDISABLE` | Disable messages with the numbers specified. |
| `--SLA` | Set the start linear address for type 5 records. |
| `--VER` | Display version and build information then quit. |
| `-ADDRESSING` | Set address fields in all `hexmate` options to use word addressing or other. |
| `-BREAK` | Break continuous data so that a new record begins at a set address. |
| `-CK` | Calculate and store a value. |
| `-FILL` | Program unused locations with a known value. |
| `-FIND` | Search and notify if a particular code sequence is detected. |
| `-FIND...,DELETE` | Remove the code sequence if it is detected. |
| `-FIND...,REPLACE` | Replace the code sequence with a new code sequence. |

**TABLE 4-2: `hexmate` COMMAND-LINE OPTIONS (CONTINUED)**

| Option | Effect |
|---|---|
| -FORMAT | Specify maximum data record length or select INHX variant. |
| -HELP | Show all options or display help message for specific option. |
| -LOGFILE | Save `hexmate` analysis of output and various results to a file. |
| -MASK | Logically AND a memory range with a bitmask. |
| -Ofile | Specify the name of the output file. |
| -SERIAL | Store a serial number or code sequence at a fixed address. |
| -SIZE | Report the number of bytes of data contained in the resultant HEX image. |
| -STRING | Store an ASCII string at a fixed address. |
| -STRPACK | Store an ASCII string at a fixed address using string packing. |
| -W | Adjust warning sensitivity. |
| + | Prefix to any option to overwrite other data in its address range, if necessary. |

If you are using the MPLAB X IDE, a log file is produced by default. It will have the project's name and the extension `.hxl`.

The input parameters to `hexmate` are now discussed in detail. The format or assumed radix of values associated with options are described with each option. Note also, that any address fields specified in these options are to be entered as byte addresses, unless specified otherwise by the `-ADDRESSING` option.

#### 4.3.1.1 SPECIFICATIONS,FILENAME.HEX

The `hexmate` application can process Intel HEX files that use either INHX32 or INHX8M format. Additional specifications can be applied to each HEX file to place restrictions or conditions on how this file should be processed.

If any specifications are used, they must precede the filename. The list of specifications will then be separated from the filename by a *comma*.

A *range restriction* can be applied with the specification `r`*`Start`*`-`*`End`*, where *`Start`* and *`End`* are both assumed to be hexadecimal values. A range restriction will cause only the address data falling within this range to be used. For example:

```
r100-1FF,myfile.hex
```

will use `myfile.hex` as input, but only process data which is addressed within the range 100h-1FFh (inclusive) from that file.

An address shift can be applied with the specification *`sOffset`*. If an address shift is used, data read from this HEX file will be shifted (by the offset specified) to a new address when generating the output. The offset can be either positive or negative. For example:

```
r100-1FFs2000,myfile.hex
```

will shift the block of data from 100h-1FFh to the new address range 2100h-21FFh.

Be careful when shifting sections of executable code. Program code should only be shifted if it is position independent.

#### 4.3.1.2 + PREFIX

When the `+` operator precedes an argument or input file, the data obtained from that source will be forced into the output file and will overwrite another other data existing at that address range. For example:

```
+input.hex +-STRING@1000=”My string”
```

Ordinarily, hexmate will issue an error if two sources try to store differing data at the same location. Using the + operator informs hexmate that if more than one data source tries to store data to the same address, the one specified with a + prefix will take priority.

#### 4.3.1.3 --EDF

This option must be used to have warning and hexmate error messages correctly displayed. The argument should be the full path to the message file to use when executing hexmate. The message files are located in the MPLAB XC8 compiler's pic/dat directory (e.g., the English language file is called en_msgs.txt).

#### 4.3.1.4 --EMAX

This option sets the maximum number of errors hexmate will display before execution is terminated, e.g., --EMAX=25. By default, up to 20 error messages will be displayed.

#### 4.3.1.5 --MSGDISABLE

This option allows error, warning or advisory messages to be disabled during execution of hexmate.

The option is passed a comma-separated list of message numbers that are to be disabled. Any error message numbers in this list are ignored unless they are followed by an :off argument. If the message list is specified as 0, then all warnings are disabled.

#### 4.3.1.6 --SLA

This option allows you to specify the linear start address for type 5 records in the Hex output file, e.g., --SLA=0x10000.

#### 4.3.1.7 --VER

This option will ask hexmate to print version, build information, and then quit.

#### 4.3.1.8 -ADDRESSING

By default, all address arguments in hexmate options expect that values will be entered as byte addresses. In some device architectures, the native addressing format can be something other than byte addressing. In these cases, it would be much simpler to be able to enter address-components in the device's native format. To facilitate this, the -ADDRESSING option is used.

This option takes one parameter that configures the number of bytes contained per address location. For example, if a device's program memory naturally used a 16-bit (2 byte) word-addressing format, the option -ADDRESSING=2 will configure hexmate to interpret all command line address fields as word addresses. The affect of this setting is global and all hexmate options will now interpret addresses according to this setting. This option will allow specification of addressing modes from one byte per address to four bytes per address.

#### 4.3.1.9 -BREAK

This option takes a *comma*-separated list of addresses. If any of these addresses are encountered in the HEX file, the current data record will conclude and a new data record will recommence from the nominated address. This can be useful to use new data records to force a distinction between functionally different areas of program space. Some HEX file readers depend on this.

#### 4.3.1.10 -CK

The -CK option is for calculating a hash value. The usage of this option is:

`-CK=start-end@dest [+offset][wWidth][tCode][gAlgorithm][pPolynomial]`

where:

- *start* and *end* specify the address range over which the hash will be calculated. If these addresses are not a multiple of the algorithm width, the value zero will be padded into the relevant input word locations that are missing.
- *dest* is the address where the hash result will be stored. This value cannot be within the range of calculation.
- *offset* is an optional initial value to be used in the calculations.
- *Width* is optional and specifies the byte-width of the result. Results can be calculated for byte-widths of 1 to 4 bytes. If a positive width is requested, the result will be stored in big-endian byte order. A negative width will cause the result to be stored in little-endian byte order. If the width is left unspecified, the result will be 2 bytes wide and stored in little-endian byte order. This width argument is not used if you have selected any Fletcher algorithm.
- *Code* is a hexadecimal code that will trail each byte in the result. This can allow each byte of the hash result to be embedded within an instruction.
- *Algorithm* is an integer to select which `hexmate` hash algorithm to use to calculate the result. A list of selectable algorithms is provided in Table 4-3. If unspecified, the default algorithm used is 8-bit checksum addition (1).
- *Polynomial* is a hexadecimal value which is the polynomial to be used if you have selected a CRC algorithm.

All numerical arguments are assumed to be hexadecimal values, except for the algorithm selector and result width, which are assumed to be decimal values.

A typical example of the use of the checksum option is:

```
-CK=0-1FFF@2FFE+2100w2g2
```

This will calculate a checksum over the range 0 to 0x1FFF and program the checksum result at address 0x2FFE. The checksum value will be offset by 0x2100. The result will be two bytes wide.

**TABLE 4-3: HEXMATE HASH ALGORITHM SELECTION**

| Selector | Algorithm Description |
|---|---|
| -5 | Reflected cyclic redundancy check (CRC) |
| -4 | Subtraction of 32 bit values from initial value |
| -3 | Subtraction of 24 bit values from initial value |
| -2 | Subtraction of 16 bit values from initial value |
| -1 | Subtraction of 8 bit values from initial value |
| 1 | Addition of 8 bit values from initial value |
| 2 | Addition of 16 bit values from initial value |
| 3 | Addition of 24 bit values from initial value |
| 4 | Addition of 32 bit values from initial value |
| 5 | Cyclic redundancy check (CRC) |
| 7 | Fletcher's checksum (8 bit calculation, 2-byte result width) |
| 8 | Fletcher's checksum (16 bit calculation, 4-byte result width) |

For more details about the algorithms that are used to calculate checksums, see Section 4.3.2 "Hash Functions".

#### 4.3.1.11 -FILL

The -FILL option is used for filling unused memory locations with a known value. The usage of this option is:

```
-FILL=[const_width:]fill_expr@address[:end_address]
```

where:

- *const_width* has the form wn and signifies the width (*n* bytes) of each constant in *fill_expr*. If *const_width* is not specified, the default value is two bytes. That is, -FILL=w1:1 with fill every unused byte with the value 0x01.
- *fill_expr* can use the syntax (where *const* and *increment* are *n*-byte constants):
  - *const* fill memory with a repeating constant; i.e., -FILL=0xBEEF becomes 0xBEEF, 0xBEEF, 0xBEEF, 0xBEEF
  - *const*+=*increment* fill memory with an incrementing constant; i.e., -FILL=0xBEEF+=1 becomes 0xBEEF, 0xBEF0, 0xBEF1, 0xBEF2
  - *const*-=*increment* fill memory with a decrementing constant; i.e., -FILL=0xBEEF-=0x10 becomes 0xBEEF, 0xBEDF, 0xBECF, 0xBEBF
  - *const*,*const*,...,*const* fill memory with a list of repeating constants; i.e., -FILL=0xDEAD,0xBEEF becomes 0xDEAD,0xBEEF,0xDEAD,0xBEEF
- The options following fill_expr result in the following behavior:
  - @*address* fill a specific address with *fill_expr*; i.e., -FILL=0xBEEF@0x1000 puts 0xBEEF at address 1000h. If the fill value is wider than the addressing value specified with -ADDRESSING, then only part of the fill value is placed in the output. For example, if the addressing is set to 1, the option above will place 0xEF at address 0x1000 and a warning will be issued.
  - @*address*:*end_address* fill a range of memory with *fill_expr*; i.e., -FILL=0xBEEF@0:0xFF puts 0xBEEF in unused addresses between 0 and 255. If the address range (multiplied by the -ADDRESSING value) is not a multiple of the fill value width, the final location will only use part of the fill value, and a warning will be issued.

The fill values are word-aligned so they start on an address that is a multiple of the fill width. Should the fill value be an instruction opcode, this alignment ensures that the instruction can be executed correctly.

All constants can be expressed in (unsigned) binary, octal, decimal or hexadecimal, as per normal C syntax, for example, 1234 is a decimal value, 0xFF00 is hexadecimal and FF00 is illegal.

#### 4.3.1.12 -FIND

This option is used to detect and log occurrences of an opcode or partial code sequence. The usage of this option is:

`-FIND=Findcode [m`*`Mask`*`]@`*`Start-End`* `[/`*`Align`*`][w][t"`*`Title`*`"]`

where:

- *`Findcode`* is the hexadecimal code sequence to search for and is entered in little endian byte order.
- *`Mask`* is optional. It specifies a bit mask applied over the *`Findcode`* value to allow a less restrictive search. It is entered in little endian byte order.
- *`Start`* and *`End`* limit the address range to search.
- *`Align`* is optional. It specifies that a code sequence can only match if it begins on an address that is a multiple of this value.
- *`w`*, if present, will cause `hexmate` to issue a warning whenever the code sequence is detected.
- *`Title`* is optional. It allows a title to be given to this code sequence. Defining a title will make log-reports and messages more descriptive and more readable. A title will not affect the actual search results.

All numerical arguments are assumed to be hexadecimal values.

Here are some examples.

The option `-FIND=3412@0-7FFF/2w` will detect the code sequence `1234h` when aligned on a `2` (two) byte address boundary, between `0h` and `7FFFh`. `w` indicates that a warning will be issued each time this sequence is found.

In this next example, `-FIND=3412M0F00@0-7FFF/2wt"ADDXY"`, the option is the same as in last example but the code sequence being matched is masked with `000Fh`, so `hexmate` will search for any of the opcodes `123xh`, where x is any digit. If a byte-mask is used, is must be of equal byte-width to the opcode it is applied to. Any messaging or reports generated by `hexmate` will refer to this opcode by the name, *ADDXY,* as this was the title defined for this search.

If `hexmate` is generating a log file, it will contain the results of all searches. `-FIND` accepts whole bytes of HEX data from 1 to 8 bytes in length. Optionally, `-FIND` can be used in conjunction with `REPLACE` or `DELETE` (as described below).

#### 4.3.1.13 -FIND...,DELETE

If the `DELETE` form of the `-FIND` option is used, any matching sequences will be removed. This function should be used with extreme caution and is not normally recommended for removal of executable code.

#### 4.3.1.14 -FIND...,REPLACE

If the `REPLACE` form of the `-FIND` option is used, any matching sequences will be replaced, or partially replaced, with new codes. The usage for this sub-option is:

`-FIND...,REPLACE=`*`Code`* `[m`*`Mask`*`]`

where:

- *`Code`* is a little endian hexadecimal code to replace the sequences that match the `-FIND` criteria.
- *`Mask`* is an optional bit mask to specify which bits within *`Code`* will replace the code sequence that has been matched. This can be useful if, for example, it is only necessary to modify 4 bits within a 16-bit instruction. The remaining 12 bits can masked and left unchanged.

#### 4.3.1.15 -FORMAT

The -FORMAT option can be used to specify a particular variant of INHX format or adjust maximum record length. The usage of this option is:

-FORMAT=*Type* [,*Length*]

where:

- *Type* specifies a particular INHX format to generate.
- *Length* is optional and sets the maximum number of bytes per data record. A valid length is between 1 and 16 decimal, with 16 being the default.

Consider the case of a bootloader trying to download an INHX32 file, which fails because it cannot process the extended address records that are part of the INHX32 standard. You know that this bootloader can only program data addressed within the range 0 to 64k, and that any data in the HEX file outside of this range can be safely disregarded. In this case, by generating the HEX file in INHX8M format the operation might succeed. The hexmate option to do this would be -FORMAT=INHX8M.

Now consider if the same bootloader also required every data record to contain exactly 8 bytes of data. This is possible by combining the -FORMAT with -FILL options. Appropriate use of -FILL can ensure that there are no gaps in the data for the address range being programmed. This will satisfy the minimum data length requirement. To set the maximum length of data records to 8 bytes, just modify the previous option to become -FORMAT=INHX8M,8.

The possible types that are supported by this option are listed in Table 4-4. Note that INHX032 is not an actual INHX format. Selection of this type generates an INHX32 file, but will also initialize the upper address information to zero. This is a requirement of some device programmers.

**TABLE 4-4: INHX TYPES USED IN -FORMAT OPTION**

| Type | Description |
|---|---|
| INHX8M | cannot program addresses beyond 64K |
| INHX32 | can program addresses beyond 64K with extended linear address records |
| INHX032 | INHX32 with initialization of upper address to zero |

#### 4.3.1.16 -HELP

Using -HELP will list all hexmate options. Entering another hexmate option as a parameter of -HELP will show a detailed help message for the given option. For example:

-HELP=string

will show additional help for the -STRING hexmate option.

#### 4.3.1.17 -LOGFILE

The -LOGFILE option saves HEX file statistics to the named file. For example:

-LOGFILE=output.hxl

will analyze the HEX file that hexmate is generating, and save a report to a file named output.hxl.

#### 4.3.1.18 -MASK

Use this option to logically AND a memory range with a particular bitmask. This is used to ensure that the unimplemented bits in program words (if any) are left blank. The usage of this option is as follows:

`-MASK=hexcode@start-end`

where *hexcode* is a value that will be ANDed with data within the *start* to *end* address range. All values are assumed to be hexadecimal. Multibyte mask values can be entered in little endian byte order.

#### 4.3.1.19 -O*FILE*

The generated Intel HEX output will be created in this file. For example:

```
-Oprogram.hex
```

will save the resultant output to `program.hex`. The output file can take the same name as one of its input files; but by doing so, it will replace the input file entirely.

#### 4.3.1.20 -SERIAL

This option will store a particular HEX value sequence at a fixed address. The usage of this option is:

`-SERIAL=Code [+/-Increment]@Address [+/-Interval][rRepetitions]`

where:

- *Code* is a hexadecimal sequence to store. The first byte specified is stored at the lowest address.
- *Increment* is optional and allows the value of *Code* to change by this value with each repetition (if requested).
- *Address* is the location to store this code, or the first repetition thereof.
- *Interval* is optional and specifies the address shift per repetition of this code.
- *Repetitions* is optional and specifies the number of times to repeat this code.

All numerical arguments are assumed to be hexadecimal values, except for the *Repetitions* argument, which is decimal value by default.

For example:

```
-SERIAL=000001@EFFE
```

will store HEX code `00001h` to address `EFFEh`.

Another example:

```
-SERIAL=0000+2@1000+10r5
```

will store 5 codes, beginning with value `0000` at address `1000h`. Subsequent codes will appear at address intervals of `+10h` and the code value will change in increments of `+2h`.

#### 4.3.1.21 -SIZE

Using the `-SIZE` option will report the number of bytes of data within the resultant HEX image to standard output. The size will also be recorded in the log file if one has been requested.

#### 4.3.1.22 -STRING

The `-STRING` option will embed an ASCII string at a fixed address. The usage of this option is:

`-STRING@`*`Address`* `[t`*`Code`*`]=”`*`Text`*`”`

where:

- *`Address`* is assumed to be a hexadecimal value representing the address at which the string will be stored.
- *`Code`* is optional and allows a byte sequence to trail each byte in the string. This can allow the bytes of the string to be encoded within an instruction.
- *`Text`* is the string to convert to ASCII and embed.

For example:

```
-STRING@1000=”My favorite string”
```

will store the ASCII data for the string, `My favorite string` (including the null character terminator), at address `1000h`.

And again:

```
-STRING@1000t34=”My favorite string”
```

will store the same string, trailing every byte in the string with the HEX code `34h`.

#### 4.3.1.23 -STRPACK

This option performs the same function as `-STRING`, but with two important differences. First, only the lower seven bits from each character are stored. Pairs of 7-bit characters are then concatenated and stored as a 14-bit word rather than in separate bytes. This is known as string packing. This is usually only useful for devices where program space is addressed as 14-bit words. The second difference is that `-STRING`'s `t` specifier is not applicable with the `-STRPACK` option.

### 4.3.2 Hash Functions

A hash value is a small fixed-size value that is calculated from, and used to represent, all the values in an arbitrary-sized block of data. If that data block is copied, a hash recalculated from the new block can be compared to the original hash. Agreement between the two hashes provides a high level of certainty that the copy is valid. There are many hash algorithms. More complex algorithms provide a more robust verification, but could use too many resources when used in an embedded environment.

`hexmate` can be used to calculate the hash of a program image that is contained in a HEX file built by the MPLAB XC8 compiler. This hash can be embedded into that HEX file and burned into the target device along with the program image. At runtime, the target device might be able to run a similar hash algorithm over the program image, now stored in its memory. If the stored and calculated hashes are the same, the embedded program can assume that it has a valid program image to execute.

`hexmate` implements several checksum and cyclic redundancy check algorithms to calculate the hash. The option to select the algorithm is described in Section 4.3.1.10 “-CK”. In the discussion of the algorithms below, it is assumed you are using the compiler driver to request a checksum or CRC.

Some consideration is required when program images contain unused memory locations. Typically unused locations should be filled with a known value to ensure consistency between results.

The following sections provide examples of the algorithms that can be used to calculate the hash at runtime.

### 4.3.3 Addition Algorithms

hexmate has several simple checksum algorithms that sum data values over a range in the program image. These algorithms correspond to the algorithm selector values 1, 2, 3, and 4, and they read the data in the program image as 1, 2, 3 or 4 byte quantities, respectively. This summation is added to an initial value (offset) that is supplied to the algorithm via the same option. The width to which the final checksum is truncated is also specified by this option and can be 1, 2, 3, or 4 bytes. hexmate will automatically store the checksum in the HEX file at the address specified in the checksum option.

Specify a hexmate option similar to the following for a 2-byte-wide checksum to be calculated from the addition of 1-byte-wide values over the address range 0x100 to 0x7fd, starting with an offset of 0x20. The checksum will be stored at 0x7fe and 0x7ff in little endian format.

```
-CK=100-7fd@7fe+20g1w-2
```

The function shown below can be customized to work with any combination of data size (readType) and checksum width (resultType).

```
typedef unsigned char readType; // size of data values read and summed
typedef unsigned int  resultType; // size of checksum result

// add to offset n additions of values starting at address data,
// truncating and returning the result
// data: the address of the first value to sum
// n:    the number of sums to perform
// offset: the intial value to which the sum is added
resultType ck_add(const readType *data, unsigned n, resultType offset)
{
        resultType chksum;

        chksum = offset;
        while(n--) {
                chksum += *data;
                data++;
        }
        return chksum;
}
```

The readType and resultType type definitions should be adjusted to suit the data read/sum width and checksum result width, respectively. When using MPLAB XC8 and for a size of 1, use a char type; for a size of 4, use a long type, etc., or consider using the exact-width types provided by <stdint.h>. If you never use an offset, that parameter can be removed and chksum assigned 0 before the loop.

hexmate can calculate a checksum over any address range; however, the test function, ck_add, assumes that the start and end address of the range being summed are a multiple of the readType width. (Clearly this is a non-issue if the size of readType is 1.) It is recommended that your checksum specification adheres to this assumption, otherwise you will need to modify the test code to perform partial reads of the starting and/or ending data values. This will significantly increase the code complexity.

### 4.3.4 Subtraction Algorithms

`hexmate` has several checksum algorithms that subtract data values over a range in the program image. These algorithms correspond to the algorithm selector values -1, -2, -3, and -4, and they read the data in the program image as 1-, 2-, 3- or 4-byte quantities, respectively. In other respects, these algorithms are identical to the addition algorithms described in Section 4.3.3 “Addition Algorithms”.

Specify a `hexmate` option similar to the following for a 4-byte-wide checksum to be calculated from the addition of 2-byte-wide values over the address range 0x0 to 0x7fd, starting with an offset of 0x0. The checksum will be stored at 0x7fe and 0x7ff in little endian format.

```
-CK=0-7fd@7feg-2w-4
```

The function shown below can be customized to work with any combination of data size (`readType`) and checksum width (`resultType`).

```
typedef unsigned char readType; // size of data values read and summed
typedef unsigned int  resultType; // size of checksum result

// add to offset n subtractions of values starting at address data,
// truncating and returning the result
// data: the address of the first value to subtract
// n:    the number of subtractions to perform
// offset: the intial value to which the subtraction is added
resultType ck_sub(const readType *data, unsigned n, resultType offset)
{
        resultType chksum;

        chksum = offset;
        while(n--) {
                chksum -= *data;
                data++;
        }
        return chksum;
}
```

### 4.3.5 Fletcher Algorithms

`hexmate` has several algorithms that implement Fletcher's checksum. These algorithms are more complex, providing a robustness approaching that of a cyclic redundancy check, but with less computational effort. There are two forms of this algorithm which correspond to the selector values 7 and 8 in the `algorithm` suboption, which also implement a 1-byte calculation and 2-byte result, as well as a 2-byte calculation and 4-byte result, respectively. `hexmate` will automatically store the checksum in the HEX file at the address specified in the checksum option.

The function shown below performs a 1-byte-wide addition and produces a 2-byte result.

```
unsigned int
fletcher8(const unsigned char * data, unsigned int n )
{
        unsigned int sum = 0xff, sumB = 0xff;
        unsigned char tlen;

        while (n) {
                tlen = n > 20 ? 20 : n;
                n -= tlen;
                do {
                        sumB += sum += *data++;
                } while (--tlen);
                sum = (sum & 0xff) + (sum >> 8);
                sumB = (sumB & 0xff) + (sumB >> 8);
        }
        sum = (sum & 0xff) + (sum >> 8);
        sumB = (sumB & 0xff) + (sumB >> 8);

        return sumB << 8 | sum;
}
```

The code for the 2-byte-addition Fletcher algorithm, producing a 4-byte result is shown below.

```
unsigned long
fletcher16(const unsigned int * data, unsigned n)
{
    unsigned long sum = 0xffff, sumB = 0xffff;
    unsigned tlen;

    while (n) {
        tlen = n > 359 ? 359 : n;
        n -= tlen;
        do {
            sumB += sum += *data++;
        } while (--tlen);
        sum = (sum & 0xffff) + (sum >> 16);
        sumB = (sumB & 0xffff) + (sumB >> 16);
    }
    sum = (sum & 0xffff) + (sum >> 16);
    sumB = (sumB & 0xffff) + (sumB >> 16);

    return sumB << 16 | sum;
}
```

### 4.3.6 CRC Algorithms

hexmate has several algorithms that implement the robust cyclic redundancy checks (CRC). There is a choice of two algorithms that correspond to the selector values 5 and -5 in the algorithm suboption, and that implement a CRC calculation and reflected CRC calculation, respectively. The reflected algorithm works on the least significant bit of the data first. The polynomial to be used and the initial value can be specified in the option. hexmate will automatically store the CRC result in the HEX file at the address specified in the checksum option.

The function shown below can be customized to work with any result width (resultType). It calculates a CRC hash value using the polynomial specified by the POLYNOMIAL macro.

```
typedef unsigned int resultType;
#define POLYNOMIAL     0x1021
#define WIDTH   (8 * sizeof(resultType))
#define MSb     ((resultType)1 << (WIDTH - 1))

resultType
crc(const unsigned char * data, unsigned n, resultType remainder) {
    unsigned pos;
    unsigned char bitp;

    for (pos = 0; pos != n; pos++) {
        remainder ^= ((resultType)data[pos] << (WIDTH - 8));
        for (bitp = 8; bitp > 0; bitp--) {
            if (remainder & MSb) {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            } else {
                remainder <<= 1;
            }
        }
    }

    return remainder;
}
```

The resultType type definition should be adjusted to suit the result width. When using MPLAB XC8 and for a size of 1, use a char type; for a size of 4, use a long type, etc., or consider using the exact-width types provided by <stdint.h>.

Here is how this function might be used when, for example, a 2-byte-wide CRC hash value is to be calculated values over the address range 0x0 to 0xFF, starting with an initial value of 0xFFFF. The result is to be stored at 0x100 and 0x101 in little endian format.

```
-CK=0-FF@100+0xFFFFg5w-2p0x1021
```

The reflected CRC result can be calculated by reflecting the input data and final result, or by reflecting the polynomial. The functions shown below can be customized to work with any result width (resultType). The crc_reflected_IO() function calculates a reflected CRC hash value by reflecting the data stream bit positions. The crc_reflected_poly() function does not adjust the data stream but reflects instead the polynomial, which in both functions is specified by the POLYNOMIAL macro. Both functions use the reflect() function to perform bit reflection.

```
typedef unsigned int resultType;
typedef unsigned char readType;
typedef unsigned int reflectWidth;

// This is the polynomial used by the CRC-16 algorithm we are using.
#define POLYNOMIAL     0x1021
```

```
#define WIDTH   (8 * sizeof(resultType))
#define MSb     ((resultType)1 << (WIDTH - 1))
#define LSb     (1)

#define REFLECT_DATA(X)        ((readType) reflect((X), 8))
#define REFLECT_REMAINDER(X)   (reflect((X), WIDTH))

reflectWidth
reflect(reflectWidth data, unsigned char nBits)
{
    reflectWidth reflection = 0;
    reflectWidth reflectMask = (reflectWidth)1 << nBits - 1;
    unsigned char bitp;

    for (bitp = 0; bitp != nBits; bitp++) {
        if (data & 0x01) {
            reflection |= reflectMask;
        }
        data >>= 1;
        reflectMask >>= 1;
    }

    return reflection;
}
```

```
resultType
crc_reflected_IO(const unsigned char * data, unsigned n, resultType
remainder) {
    unsigned pos;
    unsigned char reflected;
    unsigned char bitp;

    for (pos = 0; pos != n; pos++) {
        reflected = REFLECT_DATA(data[pos]);
        remainder ^= ((resultType)reflected << (WIDTH - 8));

        for (bitp = 8; bitp > 0; bitp--) {
            if (remainder & MSb) {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            } else {
                remainder <<= 1;
            }
        }
    }
    remainder = REFLECT_REMAINDER(remainder);

    return remainder;
}

resultType
crc_reflected_poly(const unsigned char * data, unsigned n, resultType
remainder) {
    unsigned pos;
    unsigned char bitp;
    resultType rpoly;

    rpoly = reflect(POLYNOMIAL, WIDTH);
    for (pos = 0; pos != n; pos++) {
        remainder ^= data[pos];

        for (bitp = 8; bitp > 0; bitp--) {
            if (remainder & LSb) {
                remainder = (remainder >> 1) ^ rpoly;
            } else {
                remainder >>= 1;
            }
        }
    }

    return remainder;
}
```

Here is how this function might be used when, for example, a 2-byte-wide reflected CRC result is calculated over the address range 0x0 to 0xFF, starting with an initial value of 0xFFFF. The result is to be stored at 0x100 and 0x101 in little endian format. The following option is specified when building the project. (Note the algorithm selected is *negative* 5 in this case.)

```
-CK=0-FF@100+0xFFFFg-5w-2p0x1021
```

## 4.4 OBJDUMP

The *avr-objdump* application can display various information about object files.

The general form of the tool's command line is as follows:

```
avr-objdump [options] objfiles
```

where *objfiles* can be any object file, including an archive or output file. The tool is able to determine the format of the file specified.

The `--help` option shows all the command available for `avr-objdump`.

A common usage of this tool is to obtain a full list file for the entire program. To do this, use the compiler's `-g` option when you build the project, then call the `avr-objdump` application with a command similar to the following.

```
avr-objdump.exe -S -l a.out > avr.lst
```

This will create an avr.lst listing file from the default compiler output file, showing the original C source code and line number information in the listing.

# Appendix A. Library Functions

## A.1 INTRODUCTION

The functions and preprocessor macros within the standard compiler library are alphabetically listed in this chapter.

The synopsis indicates the header file in which a declaration or definition for function or macro is found. It also shows the function prototype for functions, or the equivalent prototype for macros.

Note that where `printf()` is shown in example code, this assumes that the `putch()` function has been defined to suit the peripheral that will act as the `stdout` stream. Initialization of that peripheral must also be performed before you attempt to print.

For each built-in function for AVR, there is an equally named, uppercase built-in macro defined. That way users can easily query if or if not a specific built-in is implemented or not. For example, if __builtin_avr_nop is available the macro __BUILTIN_AVR_NOP is defined to 1 and undefined otherwise.

**TABLE A-1: DECLARATIONS PROVIDED BY <ALLOCA.H>**

| Name | Definition |
|---|---|
| `alloca` | `void * alloca (size_t __size)` |

**TABLE A-2: DECLARATIONS PROVIDED BY <ASSERT.H>**

| Name | Definition |
|---|---|
| `assert` | `void assert(scalar expression)` |

**TABLE A-3: DECLARATIONS PROVIDED BY <CTYPE.H>**

| Name | Definition |
|---|---|
| `isalnum` | `int isalnum(int c);` |
| `isalpha` | `int isalpha(int c);` |
| `isblank` | `int isblank(int c);` |
| `iscntrl` | `int iscntrl(int c);` |
| `isdigit` | `int isdigit(int c);` |
| `isgraph` | `int isgraph(int c);` |
| `islower` | `int islower(int c);` |
| `isprint` | `int isprint(int c);` |
| `ispunct` | `int ispunct(int c);` |
| `isspace` | `int isspace(int c);` |
| `isupper` | `int isupper(int c);` |
| `isxdigit` | `int isxdigit(int c);` |
| `toascii` | `int toascii (int __c)` |

**TABLE A-3: DECLARATIONS PROVIDED BY <CTYPE.H>**

| Name | Definition |
|---|---|
| tolower | int tolower(int c); |
| toupper | int toupper(int c); |

**TABLE A-4: DECLARATIONS PROVIDED BY <ERRNO.H>**

| Name | Definition |
|---|---|
| EDOM | |
| EILSEQ | |
| ERANGE | |
| errno | |

Those macros defined by <float.h>, below, that contain *XXX* are defined for float and double types, and *XXX* can be either of FLT or DBL, respectively.

**TABLE A-5: DECLARATIONS PROVIDED BY <FLOAT.H>**

| Name | Definition |
|---|---|
| FLT_RADIX | 2 |
| FLT_ROUNDS | 1 |
| FLT_EVAL_METHOD | 0 |
| DECIMAL_DIG | 9 |
| *XXX*_MAX | 3.40282346639e+38 |
| *XXX*_MIN | 1.17549435082e-38 |
| *XXX*_MIN_EXP | -125 |
| *XXX*_MIN_10_EXP | -37 |
| *XXX*_MAX_EXP | 128 |
| *XXX*_MAX_10_EXP | 38 |
| *XXX*_DIG | 6 |
| *XXX*_MANT_DIG | 24 |
| *XXX*_EPSILON | 1.19209289551e-07 |

Those macros defined by <inttypes.h>, shown below, that contain *PPP* are defined as the placeholder string for printing intmax_t and intptr_t types, and *PPP* can be either of MAX or PTR, respectively. Those macros below that contain *YYYY* are defined as the placeholder string for printing intn_t, intleastn_t and intfastn_t types (where *n* is the size, in bytes, of the type) and *YYYY* can be either of <empty>, LEAST or FAST, respectively. For example PRIdLEAST16 could be used as the placeholder string for a value with int_least16_t type.

**TABLE A-6: DECLARATIONS PROVIDED BY C99 <INTTYPES.H>**

| Name | Definition |
|---|---|
| PRId*PPP* | |
| PRId*YYYYn* | |
| PRIi*PPP* | |
| PRIi*YYYYn* | |
| PRIo*PPP* | |
| PRIo*YYYYn* | |
| PRIu*PPP* | |

**TABLE A-6: DECLARATIONS PROVIDED BY C99 <INTTYPES.H>**

| Name | Definition |
|---|---|
| PRIu*YYYYn* | |
| PRIx*PPP* | |
| PRIx*YYYYn* | |
| PRIX*PPP* | |
| PRIX*YYYYn* | |
| SCNd*PPP* | |
| SCNd*YYYYn* | |
| SCNi*PPP* | |
| SCNi*YYYYn* | |
| SCNo*PPP* | |
| SCNo*YYYYn* | |
| SCNu*PPP* | |
| SCNu*YYYYn* | |
| SCNx*PPP* | |
| SCNx*YYYYn* | |
| int_farptr_t | |
| unit_farptr_t | |

**TABLE A-7: DECLARATIONS PROVIDED BY C99 <ISO646.H>**

| Name | Definition |
|---|---|
| and | && |
| and_eq | &= |
| bitand | & |
| bitor | \| |
| compl | ~ |
| not | ! |
| not_eq | != |
| or | \|\| |
| or_eq | \|= |
| xor | ^ |
| xor_eq | ^= |

The `long long` types are 64-bit C99 Standard types when building for PIC18 devices, but when compiling to the C90 Standard or for any other device, this implementation limits their size to only 32 bits.

**TABLE A-8: DECLARATIONS PROVIDED BY <LIMITS.H>**

| Name | C99 PIC18 Definition | All other Definitions |
|---|---|---|
| CHAR_BIT | 8 | 8 |
| CHAR_MAX | 255 | 255 |
| CHAR_MIN | 0 | 0 |
| SCHAR_MAX | 127 | 127 |
| SCHAR_MIN | -128 | -128 |
| UCHAR_MAX | 255 | 255 |
| SHRT_MAX | 32767 | 32767 |
| SHRT_MIN | -32768 | -32768 |
| USHRT_MAX | 65535 | 65535 |
| INT_MAX | 32767 | 32767 |
| INT_MIN | -32768 | -32768 |
| UINT_MAX | 65535 | 65535 |
| SHRTLONG_MAX | 8388607 | 8388607 |
| SHRTLONG_MIN | -8388608 | -8388608 |
| USHRTLONG_MAX | 16777215 | 16777215 |
| LONG_MAX | 2147483647 | 2147483647 |
| LONG_MIN | -2147483648 | -2147483648 |
| ULONG_MAX | 4294967295 | 4294967295 |
| LLONG_MAX | 9223372036854775807 | 2147483647 |
| LLONG_MIN | -9223372036854775808 | -2147483648 |
| ULLONG_MAX | 18446744073709551615 | 4294967295 |

**TABLE A-9: DECLARATIONS DEFINED BY <MATH.H>**

| Name | Definition |
|---|---|
| M_E | 2.7182818284590452354 |
| M_LOG2E | 1.4426950408889634074 |
| M_LOG10E | 0.43429448190325182765 |
| M_LN2 | 0.69314718055994530942 |
| M_LN10 | 2.30258509299404568402 |
| M_PI | 3.14159265358979323846 |
| M_PI_2 | 1.57079632679489661923 |
| M_PI_4 | 0.78539816339744830962 |
| M_1_PI | 0.31830988618379067154 |
| M_2_PI | 0.63661977236758134308 |
| M_2_SQRTPI | 1.12837916709551257390 |
| M_SQRT2 | 1.41421356237309504880 |
| M_SQRT1_2 | 0.70710678118654752440 |

**TABLE A-9: DECLARATIONS DEFINED BY <MATH.H>**

| Name | Definition |
|---|---|
| INFINITY | __builtin_inf() |
| NAN | __builtin_nan("") |
| isfinite | int isfinite(*real-floating* x); |
| isinf | int isinf(*real-floating* x); |
| isnan | int isnan(*real-floating* x); |
| signbit | int signbit(*real-floating* x); |
| acos | double acos(double x); |
| acosf | float acosf(float x); |
| asin | double asin(double x); |
| asinf | float asinf(float x); |
| atan | double atan(double x); |
| atanf | float atanf(float x); |
| atan2 | double atan2(double y, double x); |
| atan2f | float atan2f(float y, float x); |
| cos | double cos(double x); |
| cosf | float cosf(float x); |
| sin | double sin(double x); |
| sinf | float sinf(float x); |
| tan | double tan(double x); |
| tanf | float tanf(float x); |
| acosh | double acosh(double x); |
| acoshf | float acoshf(float x); |
| asinh | double asinh(double x); |
| asinhf | float asinhf(float x); |
| atanh | double atanh(double x); |
| atanhf | float atanhf(float x); |
| cosh | double cosh(double x); |
| coshf | float coshf(float x); |
| sinh | double sinh(double x); |
| sinhf | float sinhf(float x); |
| tanh | double tanh(double x); |
| tanhf | float tanhf(float x); |
| exp | double exp(double x); |
| expf | float expf(float x); |
| frexp | double frexp(double value, int *exp); |
| frexpf | float frexpf(float value, int *exp); |
| ldexp | double ldexp(double x, int exp); |
| ldexpf | float ldexpf(float x, int exp); |
| log | double log(double x); |
| logf | float logf(float x); |
| log10 | double log10(double x); |
| log10f | float log10f(float x); |

**TABLE A-9: DECLARATIONS DEFINED BY <MATH.H>**

| Name | Definition |
|---|---|
| modf | double modf(double value, double *iptr); |
| modff | float modff(float value, float *iptr); |
| cbrt | double cbrt(double x); |
| cbrtf | float cbrtf(float x); |
| fabs | double fabs(double x); |
| fabsf | float fabsf(float x); |
| hypot | double hypot(double x, double y); |
| hypotf | float hypotf(float x, float y); |
| pow | double pow(double x, double y); |
| powf | float powf(float x, float y); |
| sqrt | double sqrt(double x); |
| sqrtf | float sqrtf(float x); |
| ceil | double ceil(double x); |
| ceilf | float ceilf(float x); |
| floor | double floor(double x); |
| floorf | float floorf(float x); |
| lrint | long int lrint(double x); |
| lrintf | long int lrintf(float x); |
| round | double round(double x); |
| roundf | float roundf(float x); |
| lround | long int lround(double x); |
| lroundf | long int lroundf(float x); |
| trunc | double trunc(double x); |
| truncf | float truncf(float x); |
| fmod | double fmod(double x, double y); |
| fmodf | float fmodf(float x, float y); |
| copysign | double copysign(double x, double y); |
| copysignf | float copysignf(float x, float y); |
| fdim | double fdim(double x, double y); |
| fdimf | float fdimf(float x, float y); |
| fmax | double fmax(double x, double y); |
| fmaxf | float fmaxf(float x, float y); |
| fmin | double fmin(double x, double y); |
| fminf | float fminf(float x, float y); |
| fma | double fma(double x, double y, double z); |
| fmaf | float fmaf(float x, float y, float z); |

**TABLE A-10: DECLARATIONS PROVIDED BY <SETJMP.H>**

| Name | Definition |
|---|---|
| jmp_buf | |
| setjmp | int setjmp(jmp_buf env); |

**TABLE A-10: DECLARATIONS PROVIDED BY <SETJMP.H>**

| Name | Definition |
|---|---|
| longjmp | void longjmp(jmp_buf env, int val); |

**TABLE A-11: DECLARATIONS PROVIDED BY <STDARG.H>**

| Name | Definition |
|---|---|
| va_list | |
| va_arg | *type* va_arg(va_list ap, *type*); |
| va_copy | void va_copy(va_list dest, va_list src); |
| va_end | void va_end(va_list ap); |
| va_start | void va_start(va_list ap, *parmN*); |

**TABLE A-12: DECLARATIONS PROVIDED BY C99 <STDBOOL.H>**

| Name | Definition |
|---|---|
| bool | _Bool |
| true | 1 |
| false | 0 |
| __bool_true_false_are_defined | 1 |

**TABLE A-13: DECLARATIONS PROVIDED BY <STDDEF.H>**

| Name | Definition |
|---|---|
| NULL | |
| ptrdiff_t | |
| size_t | |
| offsetof | offsetof(*type*, *member-designator*) |

Those macros defined by <stdint.h>, below, that contain *N* are defined for different sized types and *N* can be either of 8, 16, 32, or 64.

**TABLE A-14: DECLARATIONS PROVIDED BY C99 <STDINT.H>**

| Name | Definition |
|---|---|
| INT*N*_MIN | |
| INT*N*_MAX | |
| UINT*N*_MAX | |
| INT_LEAST*N*_MIN | |
| INT_LEAST*N*_MAX | |
| UINT_LEAST*N*_MAX | |
| INT_FAST*N*_MIN | |
| INT_FAST*N*_MAX | |
| UINT_FAST*N*_MAX | |
| INTPTR_MIN | |
| INTPTR_MAX | |
| UINTPTR_MAX | |
| INTMAX_MIN | |
| INTMAX_MAX | |
| UINTMAX_MAX | |
| PTRDIFF_MIN | |
| PTRDIFF_MAX | |
| SIG_ATOMIC_MIN | |
| SIG_ATOMIC_MAX | |
| SIZE_MAX | |
| INT*N*_C | INTN_C(value) |
| UINT*N*_C | UINTN_C(value) |
| INTMAX_C | INTMAX_C(value) |
| UINTMAX_C | UINTMAX_C(value) |

**TABLE A-14: DECLARATIONS PROVIDED BY C99 <STDINT.H>**

| Name | Definition |
|---|---|
| int*N*_t | |
| uint*N*_t | |
| int_least*N*_t | |
| uint_least*N*_t | |
| int_fast*N*_t | |
| uint_fast*N*_t | |
| intptr_t | |
| uintptr_t | |
| intmax_t | |
| uintmax_t | |

**TABLE A-15: DECLARATIONS PROVIDED BY <STDIO.H>**

| Name | Definition |
|---|---|
| EOF | (-1) |
| FILE | Macro |
| fpos_t | |
| putc | int putc(int c, FILE *stream); |
| putchar | int putchar(int c); |
| getc | int getc(FILE *stream); |
| getchar | int getchar(void); |
| fdev_set_udata | fdev_set_udata(stream, u) |
| fdev_get_udata | fdev_get_udata(stream) |
| fdev_setup_stream | fdev_setup_stream(stream, put, get, rwflag) |
| _FDEV_SETUP_READ | __SRD |
| _FDEV_SETUP_WRITE | __SWR |
| _FDEV_SETUP_RW | (__SRD\|__SWR) |
| _FDEV_ERR | (-1) |
| _FDEV_EOF | (-2) |
| FDEV_SETUP_STREAM | FDEV_SETUP_STREAM(put, get, rwflag) |
| size_t | |
| stderr | (__iob[2]) |
| stdin | (__iob[0]) |
| stdout | (__iob[1]) |
| fclose | int fclose(FILE *stream); |
| fflush | int fflush(FILE *stream); |
| fdevopen | FILE *fdevopen((int(*put)(char, FILE *), int(*get)(FILE *)); |
| freopen | FILE *freopen(int(*put)(char, FILE *), int(*get)(FILE *)); |
| fprintf | int fprintf(FILE * restrict stream,<br>const char * restrict format, ...); |
| fscanf | int fscanf(FILE * restrict stream,<br>const char * restrict format, ...); |
| printf | int printf(const char * restrict format, ...); |

**TABLE A-15: DECLARATIONS PROVIDED BY <STDIO.H>**

| Name | Definition |
|---|---|
| scanf | int scanf(const char * restrict format, ...); |
| snprintf | int snprintf(char * restrict s, size_t n,<br>const char * restrict format, ...); |
| sprintf | int sprintf(char * restrict s,<br>const char * restrict format, ...); |
| sscanf | int sscanf(const char * restrict s,<br>const char * restrict format, ...); |
| vfprintf | int vfprintf(FILE * restrict stream,<br>const char * restrict format, va_list arg); |
| vfscanf | int vfscanf(FILE * restrict stream,<br>const char * restrict format, va_list arg); |
| vprintf | int vprintf(const char * restrict format, va_list arg); |
| vscanf | int vscanf(const char * restrict format, va_list arg); |
| vsnprintf | int vsnprintf(char * restrict s, size_t n,<br>const char * restrict format, va_list arg); |
| vsprintf | int vsprintf(char * restrict s,<br>const char * restrict format, va_list arg); |
| vsscanf | int vsscanf(const char * restrict s,<br>const char * restrict format, va_list arg); |
| fgetc | int fgetc(FILE *stream); |
| fgets | char *fgets(char * restrict s, int n,<br>FILE * restrict stream); |
| fputc | int fputc(int c, FILE *stream); |
| fputs | int fputs(const char * restrict s,<br>FILE * restrict stream); |
| gets | char *gets(char *s); |
| puts | int puts(const char *s); |
| ungetc | int ungetc(int c, FILE *stream); |
| fread | size_t fread(void * restrict ptr, size_t size, size_t nmemb,<br>FILE * restrict stream); |
| fwrite | size_t fwrite(const void * restrict ptr,<br>size_t size, size_t nmemb,<br>FILE * restrict stream); |
| clearerr | void clearerr(FILE *stream); |
| feof | int feof(FILE *stream); |
| ferror | int ferror(FILE *stream); |

**TABLE A-16: DECLARATIONS PROVIDED BY <STDLIB.H>**

| Name | Definition |
|---|---|
| NULL | |
| EXIT_FAILURE | |
| EXIT_SUCCESS | |
| RAND_MAX | |
| size_t | |
| div_t | |
| ldiv_t | |
| RAND_MAX | |
| DTOSTR_ALWAYS_SIGN | 0x01 |
| DTOSTR_PLUS_SIGN | 0x02 |
| DTOSTR_UPPERCASE | 0x04 |
| __malloc_margin | |
| __malloc_heap_start | |
| __malloc_heap_end | |
| RANDOM_MAX | |
| ltoa | char * ltoa (long val, char *s, int radix) |
| utoa | char * utoa (unsigned int val, char *s, int radix) |
| ultoa | char * ultoa (unsigned long val, char *s, int radix) |
| itoa | char * itoa (int val, char *s, int radix) |
| random | long random (void) |
| srandom | void srandom (unsigned long __seed) |
| random_r | long random_r (unsigned long *__ctx) |
| __compar_fn_t) | (const void *, const void *) |
| atof | double atof(const char *nptr); |
| atoi | int atoi(const char *nptr); |
| atol | long int atol(const char *nptr); |
| strtod | double strtod(const char * restrict nptr,<br>char ** restrict endptr); |
| strtol | long int strtol(const char * restrict nptr,<br>char ** restrict endptr, int base); |
| strtoul | unsigned long int strtoul(const char * restrict nptr,<br>char ** restrict endptr, int base); |
| rand | int rand(void); |
| srand | void srand(unsigned int seed); |
| abort | void abort(void); |
| exit | void exit(int status); |
| bsearch | void *bsearch(const void *key, const void *base,<br>size_t nmemb, size_t size,<br>int (*compar)(const void *, const void *)); |
| qsort | void qsort(void *base, size_t nmemb, size_t size,<br>int (*compar)(const void *, const void *)); |
| abs | int abs(int j); |

**TABLE A-16: DECLARATIONS PROVIDED BY <STDLIB.H>**

| Name | Definition |
|---|---|
| labs | long int labs(long int j); |
| div | div_t div(int numer, int denom); |
| ldiv | ldiv_t ldiv(long int numer, long int denom); |
| _Exit | void _Exit(int status); |
| dtostre | char * dtostre (double __val, char *__s, unsigned char __prec, unsigned char __flags); |
| dtostrf | char * dtostrf (double __val, signed char __width, unsigned char __prec, char *__s); |

**TABLE A-17: DECLARATIONS PROVIDED BY <STRING.H>**

| | |
|---|---|
| NULL | |
| size_t | |
| ffs | int ffs (int __val); |
| ffsl | int ffsl (long __val); |
| ffsll | int ffsll (long long __val); |
| memccpy | void * memccpy (void *, const void *, int, size_t); |
| memchr | void * memchr (const void *, int, size_t) __ATTR_PURE__; |
| memcmp | int memcmp (const void *, const void *, size_t) __ATTR_PURE__; |
| memcpy | void * memcpy (void *, const void *, size_t); |
| memmem | void * memmem (const void *, size_t, const void *, size_t) __ATTR_PURE__; |
| memmove | void * memmove (void *, const void *, size_t); |
| memrchr | void * memrchr (const void *, int, size_t) __ATTR_PURE__; |
| memset | void * memset (void *, int, size_t); |
| strcat | char * strcat (char *, const char *); |
| strchr | char * strchr (const char *, int) __ATTR_PURE__; |
| strchrnul | char * strchrnul (const char *, int) __ATTR_PURE__; |
| strcmp | int strcmp (const char *, const char *) __ATTR_PURE__; |
| strcpy | char * strcpy (char *, const char *); |
| strcasecmp | int strcasecmp (const char *, const char *) __ATTR_PURE__; |
| strcasestr | char * strcasestr (const char *, const char *) __ATTR_PURE__; |
| strcspn | size_t strcspn (const char *__s, const char *__reject) __ATTR_PURE__; |
| strdup | char * strdup (const char *s1); |
| strlcat | size_t strlcat (char *, const char *, size_t); |
| strlcpy | size_t strlcpy (char *, const char *, size_t); |
| strlen | size_t strlen (const char *) __ATTR_PURE__; |
| strlwr | char * strlwr (char *); |
| strncat | char * strncat (char *, const char *, size_t); |
| strncmp | int strncmp (const char *, const char *, size_t) __ATTR_PURE__; |
| strncpy | char * strncpy (char *, const char *, size_t); |

**TABLE A-17: DECLARATIONS PROVIDED BY <STRING.H>**

| | |
|---|---|
| strncasecmp | int strncasecmp (const char *, const char *, size_t) __ATTR_PURE__; |
| strnlen | size_t strnlen (const char *, size_t) __ATTR_PURE__; |
| strpbrk | char * strpbrk (const char *__s, const char *__accept) __ATTR_PURE__; |
| strrchr | char * strrchr (const char *, int) __ATTR_PURE__; |
| strrev | char * strrev (char *); |
| strsep | char * strsep (char **, const char *); |
| strspn | size_t strspn (const char *__s, const char *__accept) __ATTR_PURE__; |
| strstr | char * strstr (const char *, const char *) __ATTR_PURE__; |
| strtok | char * strtok (char *, const char *); |
| strtok_r | char * strtok_r (char *, const char *, char **); |
| strupr | char * strupr (char *); |

**TABLE A-18: DECLARATIONS PROVIDED BY <TIME.H>**

| | |
|---|---|
| NULL | |
| ONE_HOUR | 3600 |
| ONE_DEGREE | 3600 |
| ONE_DAY | 86400 |
| UNIX_OFFSET | 946684800 |
| NTP_OFFSET | 3155673600 |
| _WEEK_DAYS_ | |
| _MONTHS_ | |
| size_t | |
| time_t | |
| tm | |
| week_date | |
| time | time_t time (time_t *timer) |
| difftime | int32_t difftime (time_t time1, time_t time0) |
| mktime | time_t mktime (struct tm *timeptr) |
| mk_gmtime | time_t mk_gmtime (const struct tm *timeptr) |
| gmtime | struct tm * gmtime (const time_t *timer) |
| gmtime_r | void gmtime_r (const time_t *timer, struct tm *timeptr) |
| localtime | struct tm * localtime (const time_t *timer) |
| localtime_r | void localtime_r (const time_t *timer, struct tm *timeptr) |
| asctime | char * asctime (const struct tm *timeptr) |
| asctime_r | void asctime_r (const struct tm *timeptr, char *buf) |
| ctime | char * ctime (const time_t *timer) |
| ctime_r | void ctime_r (const time_t *timer, char *buf) |
| isotime | char * isotime (const struct tm *tmptr) |

**TABLE A-18: DECLARATIONS PROVIDED BY <TIME.H>**

| | |
|---|---|
| isotime_r | void isotime_r (const struct tm *, char *) |
| strftime | size_t strftime (char *s, size_t maxsize, const char *format, const struct tm *timeptr) |
| set_dst | void set_dst (int(*)(const time_t *, int32_t *)) |
| set_zone | void set_zone (int32_t) |
| set_system_time | void set_system_time (time_t timestamp) |
| system_tick | void system_tick (void) |
| is_leap_year | uint8_t is_leap_year (int16_t year) |
| month_length | uint8_t month_length (int16_t year, uint8_t month) |
| week_of_year | uint8_t week_of_year (const struct tm *timeptr, uint8_t start) |
| week_of_month | uint8_t week_of_month (const struct tm *timeptr, uint8_t start) |
| week_date | struct week_date * iso_week_date (int year, int yday) |
| iso_week_date_r | void iso_week_date_r (int year, int yday, struct week_date *) |
| fatfs_time | uint32_t fatfs_time (const struct tm *timeptr) |
| set_position | void set_position (int32_t latitude, int32_t longitude) |
| equation_of_time | int16_t equation_of_time (const time_t *timer) |
| daylight_seconds | int32_t daylight_seconds (const time_t *timer) |
| solar_noon | time_t solar_noon (const time_t *timer) |
| sun_rise | time_t sun_rise (const time_t *timer) |
| sun_set | time_t sun_set (const time_t *timer) |
| solar_declination | double solar_declination (const time_t *timer) |
| moon_phase | int8_t moon_phase (const time_t *timer) |
| gm_sidereal | unsigned long gm_sidereal (const time_t *timer) |
| lm_sidereal | unsigned long lm_sidereal (const time_t *timer) |

The macros and functions provided by <xc.h> are device-specified and are described in the sections that follow Table A-19.

**TABLE A-19: DECLARATIONS PROVIDED BY <XC.H>**

| Name | Definition |
|---|---|
| di | |
| ei | |
| CRLWDT | |
| EEPROM_READ | EEPROM_READ(*address*) |
| EEPROM_WRITE | EEPROM_WRITE(*address*, *value*) |
| NOP | |
| READTIMER*X* | |
| RESET | |
| SLEEP | |
| WRITETIMER*X* | WRITETIMER(*value*) |
| eeprom_read | unsigned char eeprom_read(unsigned char address); |

**TABLE A-19: DECLARATIONS PROVIDED BY <XC.H>**

| Name | Definition |
|---|---|
| eeprom_write | void eeprom_write(unsigned char address, unsigned char value); |
| __debug_break | |
| ___mkstr | ___mkstr(*value*) |
| __EEPROM_DATA | __EEPROM_DATA(*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*) |
| get_cal_data | double get_cal_data(const unsigned char *); |
| _delay | _delay(*n*) |
| _delaywdt | _delaywdt(*n*) |
| _delay3 | _delay3(*n*) |
| __builtin_software_breakpoint | void __builtin_software_breakpoint(void); |
| __delay_ms | __delay_ms(*time*) |
| __delay_us | __delay_us(*time*) |
| __delaywdt_ms | __delaywdt_ms(*time*) |
| __delaywdt_us | __delaywdt_us(*time*) |
| __fpnormalize | double __fpnormalize(double); |
| __osccal_val | unsigned char __osccal_val(void); |

**TABLE A-20: DECLARATIONS PROVIDED BY <AVR/CPUFUNC.H>**

| Name | Definition |
|---|---|
| _NOP | |
| _MemoryBarrier | |

**TABLE A-21: DECLARATIONS PROVIDED BY <AVR/SFR_DEFS.H>**

| Name | Definition |
|---|---|
| _BV | |
| bit_is_set | bit_is_set(sfr, bit) |
| bit_is_clear | bit_is_clear(sfr, bit) |
| loop_until_bit_is_set | loop_until_bit_is_set(sfr, bit) |
| loop_until_bit_is_clear | loop_until_bit_is_clear(sfr, bit) |

**TABLE A-22: DECLARATIONS PROVIDED BY <AVR/PGMSPACE.H>**

| Name | Definition |
|---|---|
| PROGMEM | |
| PGM_P | const char * |
| PGM_VOID_P | const void * |
| PSTR(s) | ((const PROGMEM char *)(s)) |
| pgm_read_byte_near | pgm_read_byte_near(address_short) |
| pgm_read_word_near | pgm_read_word_near(address_short) |
| pgm_read_dword_near | pgm_read_dword_near(address_short) |
| pgm_read_float_near | pgm_read_float_near(address_short) |

**TABLE A-22: DECLARATIONS PROVIDED BY <AVR/PGMSPACE.H>**

| Name | Definition |
|---|---|
| pgm_read_ptr_near | pgm_read_ptr_near(address_short) |
| pgm_read_byte_far | pgm_read_byte_far(address_long) |
| pgm_read_word_far | pgm_read_word_far(address_long) |
| pgm_read_dword_far | pgm_read_dword_far(address_long) |
| pgm_read_float_far | pgm_read_float_far(address_long) |
| pgm_read_ptr_far | pgm_read_ptr_far(address_long) |
| pgm_read_byte | pgm_read_byte(address_short) |
| pgm_read_word | pgm_read_word(address_short) |
| pgm_read_dword | pgm_read_dword(address_short) |
| pgm_read_float | pgm_read_float(address_short) |
| pgm_read_ptr | pgm_read_ptr(address_short) |
| pgm_get_far_address | pgm_get_far_address(var) |
| prog_void | void PROGMEM prog_void |
| prog_char | char PROGMEM prog_char |
| prog_uchar | unsigned char PROGMEM prog_uchar |
| prog_int8_t | int8_t PROGMEM prog_int8_t |
| prog_uint8_t | uint8_t PROGMEM prog_uint8_t |
| prog_int16_t | int16_t PROGMEM prog_int16_t |
| prog_uint16_t | uint16_t PROGMEM prog_uint16_t |
| prog_int32_t | int32_t PROGMEM prog_int32_t |
| prog_uint32_t | uint32_t PROGMEM prog_uint32_t |
| prog_int64_t | int64_t PROGMEM prog_int64_t |
| prog_uint64_t | uint64_t PROGMEM prog_uint64_t |
| memchr_P | const void * memchr_P (const void *, int __val, size_t __len) |
| memcmp_P | int memcmp_P (const void *, const void *, size_t) __ATTR_PURE__ |
| memccpy_P | void * memccpy_P (void *, const void *, int __val, size_t) |
| memcpy_P | void * memcpy_P (void *, const void *, size_t) |
| memmem_P | void * memmem_P (const void *, size_t, const void *, size_t) __ATTR_PURE__ |
| memrchr_P | const void * memrchr_P (const void *, int __val, size_t __len) |
| strcat_P | char * strcat_P (char *, const char *) |
| strchr_P | const char * strchr_P (const char *, int __val) |
| strchrnul_P | const char * strchrnul_P (const char *, int __val) |
| strcmp_P | int strcmp_P (const char *, const char *) __ATTR_PURE__ |
| strcpy_P | char * strcpy_P (char *, const char *) |
| strcasecmp_P | int strcasecmp_P (const char *, const char *) __ATTR_PURE__ |
| strcasestr_P | char * strcasestr_P (const char *, const char *) __ATTR_PURE__ |

**TABLE A-22: DECLARATIONS PROVIDED BY <AVR/PGMSPACE.H>**

| Name | Definition |
|---|---|
| strcspn_P | size_t strcspn_P (const char *__s, const char *__reject) __ATTR_PURE__ |
| strlcat_P | size_t strlcat_P (char *, const char *, size_t) |
| strlcpy_P | size_t strlcpy_P (char *, const char *, size_t) |
| strnlen_P | size_t strnlen_P (const char *, size_t) |
| strncmp_P | int strncmp_P (const char *, const char *, size_t) __ATTR_PURE__ |
| strncasecmp_P | int strncasecmp_P (const char *, const char *, size_t) __ATTR_PURE__ |
| strncat_P | char * strncat_P (char *, const char *, size_t) |
| strncpy_P | char * strncpy_P (char *, const char *, size_t) |
| strpbrk_P | char * strpbrk_P (const char *__s, const char *__accept) __ATTR_PURE__ |
| strrchr_P | const char * strrchr_P (const char *, int __val) |
| strsep_P | char * strsep_P (char **__sp, const char *__delim) |
| strspn_P | size_t strspn_P (const char *__s, const char *__accept) __ATTR_PURE__ |
| strstr_P | char * strstr_P (const char *, const char *) __ATTR_PURE__ |
| strtok_P | char * strtok_P (char *__s, const char *__delim) |
| strtok_rP | char * strtok_rP (char *__s, const char *__delim, char **__last) |
| strlen_PF | size_t strlen_PF (uint_farptr_t src) |
| strnlen_PF | size_t strnlen_PF (uint_farptr_t src, size_t len) |
| memcpy_PF | void * memcpy_PF (void *dest, uint_farptr_t src, size_t len) |
| strcpy_PF | char * strcpy_PF (char *dest, uint_farptr_t src) |
| strncpy_PF | char * strncpy_PF (char *dest, uint_farptr_t src, size_t len) |
| strcat_PF | char * strcat_PF (char *dest, uint_farptr_t src) |
| strlcat_PF | size_t strlcat_PF (char *dst, uint_farptr_t src, size_t siz) |
| strncat_PF | char * strncat_PF (char *dest, uint_farptr_t src, size_t len) |
| strcmp_PF | int strcmp_PF (const char *s1, uint_farptr_t s2) __ATTR_PURE__ |
| strncmp_PF | int strncmp_PF (const char *s1, uint_farptr_t s2, size_t n) __ATTR_PURE__ |
| strcasecmp_PF | int strcasecmp_PF (const char *s1, uint_farptr_t s2) __ATTR_PURE__ |
| strncasecmp_PF | int strncasecmp_PF (const char *s1, uint_farptr_t s2, size_t n) __ATTR_PURE__ |
| strstr_PF | char * strstr_PF (const char *s1, uint_farptr_t s2) |
| strlcpy_PF | size_t strlcpy_PF (char *dst, uint_farptr_t src, size_t siz) |
| memcmp_PF | int memcmp_PF (const void *, uint_farptr_t, size_t) __ATTR_PURE__ |
| strlen_P | static size_t strlen_P (const char *s) |

**TABLE A-23: DECLARATIONS PROVIDED BY <AVR/IO.H>**

| Name | Definition |
|---|---|
| RAMEND | |
| XRAMEND | |
| E2END | |
| FLASHEND | |
| SPM_PAGESIZE | |
| E2PAGESIZE | |
| _PROTECTED_WRITE | _PROTECTED_WRITE(reg, value) |

**TABLE A-24: DECLARATIONS PROVIDED BY <AVR/BOOT.H>**

| Name | Definition |
|---|---|
| BOOTLOADER_SECTION | |
| boot_spm_interrupt_enable | |
| boot_spm_interrupt_disable | |
| boot_is_spm_interrupt | |
| boot_rww_busy | |
| boot_spm_busy | |
| boot_spm_busy_wait | |
| GET_LOW_FUSE_BITS | |
| GET_LOCK_BITS | |
| GET_EXTENDED_FUSE_BITS | |
| GET_HIGH_FUSE_BITS | |
| boot_lock_fuse_bits_get | boot_lock_fuse_bits_get(address) |
| boot_signature_byte_get | boot_signature_byte_get(addr) |
| boot_page_fill | boot_page_fill(address, data) |
| boot_page_erase | boot_page_erase(address) |
| boot_page_write | boot_page_write(address) |
| boot_rww_enable | |
| boot_lock_bits_set | boot_lock_bits_set(lock_bits) |
| boot_page_fill_safe | boot_page_fill_safe(address, data) |
| boot_page_erase_safe | boot_page_erase_safe(address) |
| boot_page_write_safe | boot_page_write_safe(address) |
| boot_rww_enable_safe | boot_rww_enable_safe() |
| boot_lock_bits_set_safe | boot_lock_bits_set_safe(lock_bits) |

**TABLE A-25: DECLARATIONS PROVIDED BY <AVR/SLEEP.H>**

| Name | Defintion |
|---|---|
| sleep_enable | void sleep_enable (void) |
| sleep_disable | void sleep_disable (void) |
| sleep_cpu | void sleep_cpu (void) |
| sleep_mode | void sleep_mode (void) |

**TABLE A-25: DECLARATIONS PROVIDED BY <AVR/SLEEP.H>**

| Name | Defintion |
|---|---|
| sleep_bod_disable | void sleep_bod_disable (void) |

**TABLE A-26: BUILTIN DECLARATIONS**

| Name | Definition |
|---|---|
| __builtin_avr_nop | void __builtin_avr_nop (void); |
| __builtin_avr_sei | void __builtin_avr_sei (void); |
| __builtin_avr_cli | void __builtin_avr_cli (void); |
| __builtin_avr_sleep | void __builtin_avr_sleep (void); |
| __builtin_avr_wdr | void __builtin_avr_wdr (void); |
| __builtin_avr_swap | unsigned char __builtin_avr_swap (unsigned char); |
| __builtin_avr_fmul | unsigned int __builtin_avr_fmul (unsigned char, unsigned char); |
| __builtin_avr_fmuls | int __builtin_avr_fmuls (char, char); |
| __builtin_avr_fmulsu | int __builtin_avr_fmulsu (char, unsigned char); |
| __builtin_avr_delay_cycles | void __builtin_avr_delay_cycles (unsigned long ticks); |
| __builtin_avr_flash_segment | char __builtin_avr_flash_segment (const __memx void*); |
| __builtin_avr_insert_bits | uint8_t __builtin_avr_insert_bits (uint32_t map, uint8_t bits, uint8_t val); |
| __builtin_avr_nops | void __builtin_avr_nops (unsigned count); |

**NOTES:**

# Appendix B. Implementation-Defined Behavior

## B.1 INTRODUCTION

This chapter indicates the compiler's choice of behavior where that behavior is implementation defined.

Items discussed in this chapter are:

- Overview
- Translation
- Environment
- Identifiers
- Characters
- Integers
- Floating-Point
- Arrays and Pointers
- Hints
- Structures, Unions, Enumerations, and Bit-Fields
- Qualifiers
- Library Functions
- Architecture

## B.2 OVERVIEW

ISO C requires a conforming implementation to document the choices for behaviors defined in the standard as "implementation-defined." The following sections list all such areas, the choices made for the compiler, and the corresponding section number from the ISO/IEC 9899:1999 (aka C99) standard or ISO/IEC 9899:1990 (aka C90).

## B.3 TRANSLATION

| | |
|---|---|
| **ISO Standard:** | "How a diagnostic is identified (3.10, 5.1.1.3)." |
| Implementation: | By default, when compiling on the command-line the following formats are used. The string `(warning)` is only displayed for warning messages. *`filename:line:column:{error/warning}: message`* |
| **ISO Standard:** | "Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2)." |
| Implementation: | The compiler will replace each leading or interleaved whitespace character sequences with a space. A trailing sequence of whitespace characters is replaced with a new-line. |

## B.4 ENVIRONMENT

| | |
|---|---|
| **ISO Standard:** | "The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2)." |
| Implementation: | Multi-byte characters are not supported in source files. |
| **ISO Standard:** | "The name and type of the function called at program start-up in a freestanding environment (5.1.2.1)." |
| Implementation: | `int main (void);` |
| **ISO Standard:** | "The effect of program termination in a freestanding environment (5.1.2.1)." |
| Implementation: | Interrupts are disabled and the programs loops indefinitely |
| **ISO Standard:** | "An alternative manner in which the `main` function may be defined (5.1.2.2.1)." |
| Implementation: | `void main (void);` |
| **ISO Standard:** | "The values given to the strings pointed to by the `argv` argument to `main` (5.1.2.2.1)." |
| Implementation: | No arguments are passed to `main`. Reference to `argc` or `argv` is undefined. |
| **ISO Standard:** | "What constitutes an interactive device (5.1.2.3)." |
| Implementation: | Application defined. |
| **ISO Standard:** | "The set of signals, their semantics, and their default handling (7.14)." |
| Implementation: | Signals are not implemented. |
| **ISO Standard:** | "Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1)." |
| Implementation: | Signals are not implemented. |
| **ISO Standard:** | "Signals for which the equivalent of `signal(`*`sig, SIG_IGN`*`);` is executed at program start-up (7.14.1.1)." |
| Implementation: | Signals are not implemented. |
| **ISO Standard:** | "The set of environment names and the method for altering the environment list used by the `getenv` function (7.20.4.5)." |
| Implementation: | The host environment is application defined. |
| **ISO Standard:** | "The manner of execution of the string by the `system` function (7.20.4.6)." |
| **Implementation:** | The host environment is application defined. |

## B.5 IDENTIFIERS

| | |
|---|---|
| **ISO Standard:** | "Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2)." |
| Implementation: | None. |
| **ISO Standard:** | "The number of significant initial characters in an identifier (5.2.4.1, 6.4.2)." |
| Implementation: | All characters are significant. |

## B.6 CHARACTERS

| | |
|---|---|
| **ISO Standard:** | "The number of bits in a byte (C90 3.4, C99 3.6)." |
| Implementation: | 8. |
| **ISO Standard:** | "The values of the members of the execution character set (C90 and C99 5.2.1)." |
| Implementation: | The execution character set is ASCII. |
| **ISO Standard:** | "The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (C90 and C99 5.2.2)." |
| Implementation: | The execution character set is ASCII. |
| **ISO Standard:** | "The value of a `char` object into which has been stored any character other than a member of the basic execution character set (C90 6.1.2.5, C99 6.2.5)." |
| Implementation: | The value of the `char` object is the 8-bit binary representation of the character in the source character set. That is, no translation is done. |
| **ISO Standard:** | "Which of `signed char` or `unsigned char` has the same range, representation, and behavior as "plain" char (C90 6.1.2.5, C90 6.2.1.1, C99 6.2.5, C99 6.3.1.1)." |
| Implementation: | By default, `signed char` is functionally equivalent to plain `char`. Id the CCI is specified, then the default is `unsigned char` The options `-funsigned-char` and `-fsigned-char` can be used to explicitly specify the type. |
| **ISO Standard:** | "The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (C90 6.1.3.4, C99 6.4.4.4, C90 and C99 5.1.1.2)." |
| Implementation: | The binary representation of the source character set is preserved to the execution character set. |
| **ISO Standard:** | "The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (C90 6.1.3.4, C99 6.4.4.4)." |
| Implementation: | The previous value is shifted left by eight, and the bit pattern of the next character is masked in. The final result is of type `int`. If the result is larger than can be represented by an `int`, a warning diagnostic is issued and the value truncated to `int` size. |
| **ISO Standard:** | "The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (C90 6.1.3.4, C99 6.4.4.4)." |
| Implementation: | Multi-byte characters are not supported in source files. |
| **ISO Standard:** | "The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (C90 6.1.3.4, C99 6.4.4.4)." |
| Implementation: | Multi-byte characters are not supported in source files. |
| **ISO Standard:** | "The current locale used to convert a wide string literal into corresponding wide character codes (C90 6.1.4, C99 6.4.5)." |
| Implementation: | Wide strings are not supported. |
| **ISO Standard:** | "The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (C90 6.1.4, C99 6.4.5)." |
| Implementation: | Multi-byte characters are not supported in source files. |

## B.7 INTEGERS

| | |
|---|---|
| **ISO Standard:** | "Any extended integer types that exist in the implementation (C99 6.2.5)." |
| Implementation: | The `__int24` and `__uint24` keywords designate a signed and unsigned, respectively, 24-bit integer type. |
| **ISO Standard:** | "Whether signed integer types are represented using sign and magnitude, two's complement, or one's complement, and whether the extraordinary value is a trap representation or an ordinary value (C99 6.2.6.2)." |
| Implementation: | All integer types are represented as two's complement, and all bit patterns are ordinary values. |
| **ISO Standard:** | "The rank of any extended integer type relative to another extended integer type with the same precision (C99 6.3.1.1)." |
| Implementation: | There are no extended integer types with the same precision. |
| **ISO Standard:** | "The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (C90 6.2.1.2, C99 6.3.1.3)." |
| Implementation: | When converting value X to a type of width N, the value of the result is the Least Significant N bits of the 2's complement representation of X. That is, X is truncated to N bits. No signal is raised. |
| **ISO Standard:** | "The results of some bitwise operations on signed integers (C90 6.3, C99 6.5)." |
| Implementation: | The right shift operator sign extends signed values. Thus, an object with the signed int value 0x0124 shifted right one bit will yield the value 0x0092 and the value 0x8024 shifted right one bit will yield the value 0xC012. Right shifts of unsigned integral values always clear the MSb of the result. Left shifts (<< operator), signed or unsigned, always clear the LSb of the result.<br>Other bitwise operations act as if the operand was unsigned. |

## B.8 FLOATING-POINT

| | |
|---|---|
| **ISO Standard:** | "The accuracy of the floating-point operations and of the library functions in <math.h> and <complex.h> that return floating-point results (C90 and C99 5.2.4.2.2)." |
| Implementation: | The accuracy is unknown. |
| **ISO Standard:** | "The rounding behaviors characterized by non-standard values of FLT_ROUNDS (C90 and C99 5.2.4.2.2)." |
| Implementation: | No such values are used. |
| **ISO Standard:** | "The evaluation methods characterized by non-standard negative values of FLT_EVAL_METHOD (C90 and C99 5.2.4.2.2)." |
| Implementation: | No such values are used. |
| **ISO Standard:** | "The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (C90 6.2.1.3, C99 6.3.1.4)." |
| Implementation: | The integer is rounded to the nearest floating point representation. |
| **ISO Standard:** | "The direction of rounding when a floating-point number is converted to a narrower floating-point number (C90 6.2.1.4, 6.3.1.5)." |
| Implementation: | A floating-point number is rounded down when converted to a narrow floating-point value. |
| **ISO Standard:** | "How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (C90 6.1.3.1, C99 6.4.4.2)." |
| Implementation: | Not applicable; FLT_RADIX is a power of 2. |
| **ISO Standard:** | "Whether and how floating expressions are contracted when not disallowed by the FP_CONTRACT pragma (C99 6.5)." |
| Implementation: | The pragma is not implemented. |
| **ISO Standard:** | "The default state for the FENV_ACCESS pragma (C99 7.6.1)." |
| Implementation: | This pragma is not implemented. |
| **ISO Standard:** | "Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (C99 7.6, 7.12)." |
| Implementation: | None supported. |
| **ISO Standard:** | "The default state for the FP_CONTRACT pragma (C99 7.12.2)." |
| Implementation: | This pragma is not implemented. |
| **ISO Standard:** | "Whether the "inexact" floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (C99 F.9)." |
| Implementation: | The exception is not raised. |
| **ISO Standard:** | "Whether the "underflow" (and "inexact") floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (C99 F.9)." |
| Implementation: | The exception is not raised. |

## B.9 ARRAYS AND POINTERS

| | |
|---|---|
| **ISO Standard:** | "The result of converting a pointer to an integer or vice versa (C90 6.3.4, C99 6.3.2.3)." |
| Implementation: | A cast from an integer to a pointer or vice versa results uses the binary representation of the source type, reinterpreted as appropriate for the destination type.<br>If the source type is larger than the destination type, the most significant bits are discarded. When casting from a pointer to an integer, if the source type is smaller than the destination type, the result is sign extended. When casting from an integer to a pointer, if the source type is smaller than the destination type, the result is extended based on the signedness of the source type. |
| **ISO Standard:** | "The size of the result of subtracting two pointers to elements of the same array (C90 6.3.6, C99 6.5.6)." |
| Implementation: | The signed integer result will have the same size as the pointer operands in the subtraction. |

## B.10 HINTS

| | |
|---|---|
| **ISO Standard:** | "The extent to which suggestions made by using the `register` storage-class specifier are effective (C90 6.5.1, C99 6.7.1)." |
| Implementation: | The register storage class can be used to locate certain objects in a register (see Section 3.5.6 "Variables in Registers"). |
| **ISO Standard:** | "The extent to which suggestions made by using the inline function specifier are effective (C99 6.7.4)." |
| Implementation: | A function might be inlined if a PRO-licensed compiler has the optimizers set to level 2 or higher. In other situations, the function will not be inlined. |

## B.11 STRUCTURES, UNIONS, ENUMERATIONS, AND BIT-FIELDS

| | |
|---|---|
| **ISO Standard:** | "Whether a "plain" `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field (C90 6.5.2, C90 6.5.2.1, C99 6.7.2, C99 6.7.2.1)." |
| Implementation: | A plain `int` bit-field is treated as an unsigned integer. The `-fsigned-bitfields` option can be used to treat bit-fields as signed. |
| **ISO Standard:** | "Allowable bit-field types other than `_Bool`, `signed int`, and `unsigned int` (C99 6.7.2.1)." |
| Implementation: | All integer types are allowed. |
| **ISO Standard:** | "Whether a bit-field can straddle a storage unit boundary (C90 6.5.2.1, C99 6.7.2.1)." |
| Implementation: | A bit-field can straddle a storage unit. |
| **ISO Standard:** | "The order of allocation of bit-fields within a unit (C90 6.5.2.1, C99 6.7.2.1)." |
| Implementation: | The first bit-field defined in a structure is allocated the LSb position in the storage unit. Subsequent bit-fields are allocated higher-order bits. |
| **ISO Standard:** | "The alignment of non-bit-field members of structures (C90 6.5.2.1, C99 6.7.2.1)." |
| Implementation: | No alignment is performed. |
| **ISO Standard:** | "The integer type compatible with each enumerated type (C90 6.5.2.2, C99 6.7.2.2)." |
| Implementation: | A signed or unsigned `int` can be chosen to represent an enumerated type. |

## B.12 QUALIFIERS

| | |
|---|---|
| **ISO Standard:** | “What constitutes an access to an object that has `volatile`-qualified type (C90 6.5.3, C99 6.7.3).” |
| Implementation: | Each reference to the identifier of a `volatile`-qualified object constitutes one access to the object. |

## B.13 PRE-PROCESSING DIRECTIVES

| | |
|---|---|
| **ISO Standard:** | “How sequences in both forms of header names are mapped to headers or external source file names (C90 6.1.7, C99 6.4.7).” |
| Implementation: | The character sequence between the delimiters is considered to be a string which is a file name for the host environment. |
| **ISO Standard:** | “Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (C90 6.8.1, C99 6.10.1).” |
| Implementation: | Yes. |
| **ISO Standard:** | “Whether the value of a single-character `character` constant in a constant expression that controls conditional inclusion may have a negative value (C90 6.8.1, C99 6.10.1).” |
| Implementation: | Yes. |
| **ISO Standard:** | “The places that are searched for an included `< >` delimited header, and how the places are specified or the header is identified (C90 6.8.2, C99 6.10.2).” |
| Implementation: | The preprocessor searches any directory specified using the `-I` option, then, provided the `-nostdinc` option has not been used, the standard compiler include directory, `<install directory>/avr/avr/include`. |
| **ISO Standard:** | “How the named source file is searched for in an included `""` delimited header (C90 6.8.2, C99 6.10.2).” |
| Implementation: | The compiler first searches for the named file in the directory containing the including file, then the directories which are searched for a `< >` delimited header. |
| **ISO Standard:** | “The method by which preprocessing tokens are combined into a header name (C90 6.8.2, C99 6.10.2).” |
| Implementation: | All tokens, including whitespace, are considered part of the header file name. Macro expansion is not performed on tokens inside the delimiters. |
| **ISO Standard:** | “The nesting limit for `#include` processing (C90 6.8.2, C99 6.10.2).” |
| Implementation: | No limit. |
| **ISO Standard:** | "Whether the # operator inserts a \ character before the \ character that begins a universal character name in a character constant or string literal (6.10.3.2)." |
| Implementation: | No. |
| **ISO Standard:** | “The behavior on each recognized non-`STDC #pragma` directive (C90 6.8.6, C99 6.10.6).” |
| Implementation: | See Section 3.14.3 “Pragma Directives” |
| **ISO Standard:** | “The definitions for `__DATE__` and `__TIME__` when respectively, the date and time of translation are not available (C90 6.8.8, C99 6.10.8).” |
| Implementation: | The date and time of translation are always available. |

## B.14 LIBRARY FUNCTIONS

| | |
|---|---|
| **ISO Standard:** | "Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1)." |
| Implementation: | See Appendix A. Library Functions. |
| **ISO Standard:** | "The format of the diagnostic printed by the `assert` macro (7.2.1.1)." |
| Implementation: | `Assertion failed: (`*`message`*`), function `*`function`*`, file `*`file`*`, line `*`line`*`.\n"`. The `function` *`function`* component is skipped if `__func__` is unavailable. |
| **ISO Standard:** | "The representation of floating-point exception flags stored by the `fegetexceptflag` function (7.6.2.2)." |
| **I**Implementation: | Unimplemented. |
| **ISO Standard:** | "Whether the `feraiseexcept` function raises the inexact exception in addition to the overflow or underflow exception (7.6.2.3)." |
| Implementation: | Unimplemented. |
| **ISO Standard:** | "Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function (7.11.1.1)." |
| Implementation: | None. |
| **ISO Standard:** | "The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 or greater than 2 (7.12)." |
| Implementation: | Unimplemented. |
| **ISO Standard:** | "Domain errors for the mathematics functions, other than those required by this International Standard (7.12.1)." |
| Implementation: | None. |
| **ISO Standard:** | "The values returned by the mathematics functions on domain errors (7.12.1)." |
| Implementation: | `errno` is set to `EDOM` on domain errors. |
| **ISO Standard:** | "Whether the mathematics functions set `errno` to the value of the macro `ERANGE` on overflow and/or underflow range errors (7.12.1)." |
| Implementation: | Yes |
| **ISO Standard:** | "Whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero (7.12.10.1)." |
| Implementation: | The first argument is returned. |
| **ISO Standard:** | "The base-2 logarithm of the modulus used by the `remquo` function in reducing the quotient (7.12.10.3)." |
| Implementation: | Unimplemented. |
| **ISO Standard:** | Whether the equivalent of signal(sig, SIG_DFL); is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1). |
| Implementation: | Signals are not implemented. |
| **ISO Standard:** | The null pointer constant to which the macro `NULL` expands (7.17). |
| Implementation: | `((void *)0)` |
| **ISO Standard:** | "Whether the last line of a text stream requires a terminating new-line character (7.19.2)." |
| Implementation: | Streams are not implemented. |
| **ISO Standard:** | "Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.19.2)." |
| Implementation: | Streams are not implemented. |
| **ISO Standard:** | "The number of null characters that may be appended to data written to a binary stream (7.19.2)." |
| Implementation: | Streams are not implemented. |

| | |
|---|---|
| **ISO Standard:** | "Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.19.3)." |
| Implementation: | Streams are not implemented. |
| **ISO Standard:** | "Whether a write on a text stream causes the associated file to be truncated beyond that point (7.19.3)." |
| Implementation: | Streams are not implemented. |
| **ISO Standard:** | "The characteristics of file buffering (7.19.3)." |
| Implementation: | File handling is not implemented. |
| **ISO Standard:** | "Whether a zero-length file actually exists (7.19.3)." |
| Implementation: | File handling is not implemented. |
| **ISO Standard:** | "The rules for composing valid file names (7.19.3)." |
| Implementation: | File handling is not implemented. |
| **ISO Standard:** | "Whether the same file can be open multiple times (7.19.3)." |
| Implementation: | File handling is not implemented. |
| **ISO Standard:** | "The nature and choice of encodings used for multibyte characters in files (7.19.3)." |
| Implementation: | File handling is not implemented. |
| **ISO Standard:** | "The effect of the `remove` function on an open file (7.19.4.1)." |
| Implementation: | File handling is not implemented. |
| **ISO Standard:** | "The effect if a file with the new name exists prior to a call to the `rename` function (7.19.4.2)." |
| Implementation: | File handling is not implemented. |
| **ISO Standard:** | "Whether an open temporary file is removed upon abnormal program termination (7.19.4.3)." |
| Implementation: | File handling is not implemented. |
| **ISO Standard:** | "What happens when the `tmpnam` function is called more than `TMP_MAX` times (7.19.4.4)." |
| Implementation: | File handling is not implemented. |
| **ISO Standard:** | "Which changes of mode are permitted (if any), and under what circumstances (7.19.5.4)." |
| Implementation: | File handling is not implemented. |
| **ISO Standard:** | "The style used to print an infinity or NaN and the meaning of the *n-char-sequence,* if that style is printed for a NaN (7.19.6.1, 7.24.2.1)." |
| Implementation: | The values are printed as the nearest number. |
| **ISO Standard:** | "The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.19.6.1, 7.24.2.1)." |
| Implementation: | Functionally equivalent to `%lx`. |
| **ISO Standard:** | "The interpretation of a `-` character that is neither the first nor the last character, nor the second where a `^` character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.2)." |
| Implementation: | Streams are not implemented. |
| **ISO Standard:** | "The set of sequences matched by the `%p` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.2)." |
| Implementation: | Streams are not implemented. |
| **ISO Standard:** | "The value to which the macro `errno` is set by the `fgetpos`, `fsetpos`, or `ftell` functions on failure (7.19.9.1, 7.19.9.3, 7.19.9.4)." |
| Implementation: | Streams are not implemented. |
| **ISO Standard:** | "The meaning of the *n-char-sequence* in a string converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function (7.20.1.3, 7.24.4.1.1)." |

| | |
|---|---|
| Implementation: | No meaning is attached to the sequence. |
| **ISO Standard:** | "Whether or not the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function sets `errno` to `ERANGE` when underflow occurs (7.20.1.3, 7.24.4.1.1)." |
| Implementation: | No. |
| **ISO Standard:** | "Whether the `calloc`, `malloc`, and `realloc` functions return a Null Pointer or a pointer to an allocated object when the size requested is zero (7.20.3)." |
| Implementation: | The requested size is bumped to the lowest allowable, two bytes. If this can be successfully allocated, a pointer to the space is returned; otherwise `NULL` is returned. |
| **ISO Standard:** | "Whether open output streams are flushed, open streams are closed, or temporary files are removed when the `abort` function is called (7.20.4.1)." |
| Implementation: | Streams are not implemented. |
| **ISO Standard:** | "The termination status returned to the host environment by the `abort` function (7.20.4.1)." |
| Implementation: | The host environment is application defined. |
| **ISO Standard:** | "The value returned by the `system` function when its argument is not a Null Pointer (7.20.4.5)." |
| Implementation: | The host environment is application defined. |
| **ISO Standard:** | "The local time zone and Daylight Saving Time (7.23.1)." |
| Implementation: | Application defined. |
| **ISO Standard:** | "The range and precision of times representable in `clock_t` and `time_t` (7.23)" |
| Implementation: | the `time_t` type is used to hold a number of seconds and is defined as a `long` type; `clock_t` is not defined. |
| **ISO Standard:** | "The era for the `clock` function (7.23.2.1)." |
| Implementation: | Application defined. |
| **ISO Standard:** | "The replacement string for the `%Z` specifier to the `strftime`, `strfxtime`, `wcsftime`, and `wcsfxtime` functions in the "`C`" locale (7.23.3.5, 7.23.3.6, 7.24.5.1, 7.24.5.2)." |
| Implementation: | These functions are unimplemented. |
| **ISO Standard:** | "Whether or when the trigonometric, hyperbolic, base-*e* exponential, base-*e* logarithmic, error, and log gamma functions raise the inexact exception in an IEC 60559 conformant implementation (F.9)." |
| Implementation: | No. |
| **ISO Standard:** | "Whether the functions in `<math.h>` honor the Rounding Direction mode (F.9)." |
| Implementation: | The rounding mode is not forced. |

## B.15 ARCHITECTURE

| | |
|---|---|
| **ISO Standard:** | "The values or expressions assigned to the macros specified in the headers `<float.h>`, `<limits.h>`, and `<stdint.h>` (C90 and C99 5.2.4.2, C99 7.18.2, 7.18.3)." |
| Implementation: | See Table A-5, Table A-8 and the header files in `<install directory>/avr/avr/include/c99`. |
| **ISO Standard:** | "The number, order, and encoding of bytes in any object, when not explicitly specified in the standard (C99 6.2.6.1)." |
| Implementation: | Little endian, populated from Least Significant Byte first. |
| **ISO Standard:** | "The value of the result of the `sizeof` operator (C90 6.3.3.4, C99 6.5.3.4)." |
| Implementation: | The type of the result is equivalent to `unsigned int`. |

**NOTES:**

# Glossary

## A

**Absolute Section**

A GCC compiler section with a fixed (absolute) address that cannot be changed by the linker.

**Absolute Variable/Function**

A variable or function placed at an absolute address using the OCG compiler's `@ address` syntax.

**Access Memory**

PIC18 Only – Special registers on PIC18 devices that allow access regardless of the setting of the Bank Select Register (BSR).

**Access Entry Points**

Access entry points provide a way to transfer control across segments to a function which cannot be defined at link time. They support the separate linking of boot and secure application segments.

**Address**

Value that identifies a location in memory.

**Alphabetic Character**

Alphabetic characters are those characters that are letters of the Arabic alphabet (a, b, …, z, A, B, …, Z).

**Alphanumeric**

Alphanumeric characters are comprised of alphabetic characters and decimal digits (0,1, …, 9).

**ANDed Breakpoints**

Set up an ANDed condition for breaking, i.e., breakpoint 1 AND breakpoint 2 must occur at the same time before a program halt. This can only be accomplished if a data breakpoint and a program memory breakpoint occur at the same time.

**Anonymous Structure**

16-bit C Compiler **–** An unnamed structure.

PIC18 C Compiler **–** An unnamed structure that is a member of a C union. The members of an anonymous structure can be accessed as if they were members of the enclosing union. For example, in the following code, `hi` and `lo` are members of an anonymous structure inside the union `caster`.

```
union castaway {
 int intval;
 struct {
  char lo; //accessible as caster.lo
  char hi; //accessible as caster.hi
 };
} caster;
```

**ANSI**

American National Standards Institute is an organization responsible for formulating and approving standards in the United States.

**Application**

A set of software and hardware that can be controlled by a PIC microcontroller.

**Archive/Archiver**

An archive/library is a collection of relocatable object modules. It is created by assembling multiple source files to object files, and then using the archiver/librarian to combine the object files into one archive/library file. An archive/library can be linked with object modules and other archives/libraries to create executable code.

**ASCII**

American Standard Code for Information Interchange is a character set encoding that uses 7 binary digits to represent each character. It includes upper and lower case letters, digits, symbols and control characters.

**Assembly/Assembler**

Assembly is a programming language that describes binary machine code in a symbolic form. An assembler is a language tool that translates assembly language source code into machine code.

**Assigned Section**

A GCC compiler section which has been assigned to a target memory block in the linker command file.

**Asynchronously**

Multiple events that do not occur at the same time. This is generally used to refer to interrupts that can occur at any time during processor execution.

**Asynchronous Stimulus**

Data generated to simulate external inputs to a simulator device.

**Attribute**

GCC characteristics of variables or functions in a C program which are used to describe machine-specific properties.

**Attribute, Section**

GCC characteristics of sections, such as "executable", "readonly", or "data" that can be specified as flags in the assembler `.section` directive.

## B

**Binary**

The base two numbering system that uses the digits 0-1. The rightmost digit counts ones, the next counts multiples of 2, then $2^2 = 4$, etc.

**Breakpoint**

Hardware Breakpoint: An event whose execution will cause a halt.

Software Breakpoint: An address where execution of the firmware will halt. Usually achieved by a special break instruction.

**Build**

Compile and link all the source files for an application.

## C

**C\C++**

C is a general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators. C++ is the object-oriented version of C.

**Calibration Memory**

A special function register or registers used to hold values for calibration of a PIC microcontroller on-board RC oscillator or other device peripherals.

**Central Processing Unit**

The part of a device that is responsible for fetching the correct instruction for execution, decoding that instruction, and then executing that instruction. When necessary, it works in conjunction with the arithmetic logic unit (ALU) to complete the execution of the instruction. It controls the program memory address bus, the data memory address bus, and accesses to the stack.

**Clean**

Clean removes all intermediary project files, such as object, hex and debug files, for the active project. These files are recreated from other files when a project is built.

**COFF**

Common Object File Format. An object file of this format contains machine code, debugging and other information.

**Command Line Interface**

A means of communication between a program and its user based solely on textual input and output.

**Compiled Stack**

A region of memory managed by the compiler in which variables are statically allocated space. It replaces a software or hardware stack when such mechanisms cannot be efficiently implemented on the target device.

**Compiler**

A program that translates a source file written in a high-level language into machine code.

**Conditional Assembly**

Assembly language code that is included or omitted based on the assembly-time value of a specified expression.

**Conditional Compilation**

The act of compiling a program fragment only if a certain constant expression, specified by a preprocessor directive, is true.

**Configuration Bits**

Special-purpose bits programmed to set PIC microcontroller modes of operation. A Configuration bit can or cannot be preprogrammed.

**Control Directives**

Directives in assembly language code that cause code to be included or omitted based on the assembly-time value of a specified expression.

**CPU**

*See* Central Processing Unit.

**Cross Reference File**

A file that references a table of symbols and a list of files that references the symbol. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

## D

**Data Directives**

Data directives are those that control the assembler's allocation of program or data memory and provide a way to refer to data items symbolically; that is, by meaningful names.

**Data Memory**

On Microchip MCU and DSC devices, data memory (RAM) is comprised of General Purpose Registers (GPRs) and Special Function Registers (SFRs). Some devices also have EEPROM data memory.

**Data Monitor and Control Interface (DMCI)**

The Data Monitor and Control Interface, or DMCI, is a tool in MPLAB X IDE. The interface provides dynamic input control of application variables in projects. Application-generated data can be viewed graphically using any of 4 dynamically-assignable graph windows.

**Debug/Debugger**

*See* ICE/ICD.

**Debugging Information**

Compiler and assembler options that, when selected, provide varying degrees of information used to debug application code. See compiler or assembler documentation for details on selecting debug options.

**Deprecated Features**

Features that are still supported for legacy reasons, but will eventually be phased out and no longer used.

**Device Programmer**

A tool used to program electrically programmable semiconductor devices such as microcontrollers.

**Digital Signal Controller**

A A digital signal controller (DSC) is a microcontroller device with digital signal processing capability, i.e., Microchip dsPIC DSC devices.

**Digital Signal Processing\Digital Signal Processor**

Digital signal processing (DSP) is the computer manipulation of digital signals, commonly analog signals (sound or image) which have been converted to digital form (sampled). A digital signal processor is a microprocessor that is designed for use in digital signal processing.

**Directives**

Statements in source code that provide control of the language tool's operation.

**Download**

Download is the process of sending data from a host to another device, such as an emulator, programmer or target board.

**DWARF**

Debug With Arbitrary Record Format. DWARF is a debug information format for ELF files.

## E

**EEPROM**

Electrically Erasable Programmable Read Only Memory. A special type of PROM that can be erased electrically. Data is written or erased one byte at a time. EEPROM retains its contents even when power is turned off.

**ELF**

Executable and Linking Format. An object file of this format contains machine code. Debugging and other information is specified in with DWARF. ELF/DWARF provide better debugging of optimized code than COFF.

**Emulation/Emulator**

*See* ICE/ICD.

**Endianness**

The ordering of bytes in a multi-byte object.

**Environment**

MPLAB PM3 **–** A folder containing files on how to program a device. This folder can be transferred to a SD/MMC card.

**Epilogue**

A portion of compiler-generated code that is responsible for deallocating stack space, restoring registers and performing any other machine-specific requirement specified in the runtime model. This code executes after any user code for a given function, immediately prior to the function return.

**EPROM**

Erasable Programmable Read Only Memory. A programmable read-only memory that can be erased usually by exposure to ultraviolet radiation.

**Error/Error File**

An error reports a problem that makes it impossible to continue processing your program. When possible, an error identifies the source file name and line number where the problem is apparent. An error file contains error messages and diagnostics generated by a language tool.

**Event**

A description of a bus cycle which can include address, data, pass count, external input, cycle type (fetch, R/W), and time stamp. Events are used to describe triggers, breakpoints and interrupts.

**Executable Code**

Software that is ready to be loaded for execution.

**Export**

Send data out of the MPLAB IDE in a standardized format.

**Expressions**

Combinations of constants and/or symbols separated by arithmetic or logical operators.

**Extended Microcontroller Mode**

In extended microcontroller mode, on-chip program memory as well as external memory is available. Execution automatically switches to external if the program memory address is greater than the internal memory space of the PIC18 device.

**Extended Mode (PIC18 MCUs)**

In Extended mode, the compiler will utilize the extended instructions (i.e., `ADDFSR`, `ADDULNK`, `CALLW`, `MOVSF`, `MOVSS`, `PUSHL`, `SUBFSR` and `SUBULNK`) and the indexed with literal offset addressing.

**External Label**

A label that has external linkage.

**External Linkage**

A function or variable has external linkage if it can be referenced from outside the module in which it is defined.

**External Symbol**

A symbol for an identifier which has external linkage. This can be a reference or a definition.

**External Symbol Resolution**

A process performed by the linker in which external symbol definitions from all input modules are collected in an attempt to resolve all external symbol references. Any external symbol references which do not have a corresponding definition cause a linker error to be reported.

**External Input Line**

An external input signal logic probe line (TRIGIN) for setting an event based upon external signals.

**External RAM**

Off-chip Read/Write memory.

## F

**Fatal Error**

An error that will halt compilation immediately. No further messages will be produced.

**File Registers**

On-chip data memory, including General Purpose Registers (GPRs) and Special Function Registers (SFRs).

**Filter**

Determine by selection what data is included/excluded in a trace display or data file.

**Fixup**

The process of replacing object file symbolic references with absolute addresses after relocation by the linker.

**Flash**

A type of EEPROM where data is written or erased in blocks instead of bytes.

**FNOP**

Forced No Operation. A forced NOP cycle is the second cycle of a two-cycle instruction. Because the PIC microcontroller architecture is pipelined, it prefetches the next instruction in the physical address space while it is executing the current instruction. However, if the current instruction changes the program counter, this prefetched instruction is explicitly ignored, causing a forced NOP cycle.

**Frame Pointer**

A pointer that references the location on the stack that separates the stack-based arguments from the stack-based local variables. Provides a convenient base from which to access local variables and other values for the current function.

**Free-Standing**

An implementation that accepts any strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (ANSI ‘89 standard clause 7) is confined to the contents of the standard headers `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>` and `<stdint.h>`.

## G

**GPR**

General Purpose Register. The portion of device data memory (RAM) available for general use.

## H

**Halt**

A stop of program execution. Executing Halt is the same as stopping at a breakpoint.

**Heap**

An area of memory used for dynamic memory allocation where blocks of memory are allocated and freed in an arbitrary order determined at runtime.

**Hex Code\Hex File**

Hex code is executable instructions stored in a hexadecimal format code. Hex code is contained in a hex file.

**Hexadecimal**

The base 16 numbering system that uses the digits 0-9 plus the letters A-F (or a-f). The digits A-F represent hexadecimal digits with values of (decimal) 10 to 15. The rightmost digit counts ones, the next counts multiples of 16, then $16^2$ = 256, etc.

**High Level Language**

A language for writing programs that is further removed from the processor than assembly.

## I

**ICE/ICD**

In-Circuit Emulator/In-Circuit Debugger: A hardware tool that debugs and programs a target device. An emulator has more features than an debugger, such as trace.

In-Circuit Emulation/In-Circuit Debug: The act of emulating or debugging with an in-circuit emulator or debugger.

-ICE/-ICD: A device (MCU or DSC) with on-board in-circuit emulation or debug circuitry. This device is always mounted on a header board and used to debug with an in-circuit emulator or debugger.

**ICSP**

In-Circuit Serial Programming. A method of programming Microchip embedded devices using serial communication and a minimum number of device pins.

**IDE**

Integrated Development Environment, as in MPLAB IDE.

**Identifier**

A function or variable name.

**IEEE**

Institute of Electrical and Electronics Engineers.

**Import**

Bring data into the MPLAB IDE from an outside source, such as from a hex file.

**Initialized Data**

Data which is defined with an initial value. In C,

```
int myVar=5;
```

defines a variable which will reside in an initialized data section.

**Instruction Set**

The collection of machine language instructions that a particular processor understands.

**Instructions**

A sequence of bits that tells a central processing unit to perform a particular operation and can contain data to be used in the operation.

**Internal Linkage**

A function or variable has internal linkage if it cannot be accessed from outside the module in which it is defined.

**International Organization for Standardization**

An organization that sets standards in many businesses and technologies, including computing and communications. Also known as ISO.

**Interrupt**

A signal to the CPU that suspends the execution of a running application and transfers control to an Interrupt Service Routine (ISR) so that the event can be processed. Upon completion of the ISR, normal execution of the application resumes.

**Interrupt Handler**

A routine that processes special code when an interrupt occurs.

**Interrupt Service Request (IRQ)**

An event which causes the processor to temporarily suspend normal instruction execution and to start executing an interrupt handler routine. Some processors have several interrupt request events allowing different priority interrupts.

**Interrupt Service Routine (ISR)**

Language tools **–** A function that handles an interrupt.

MPLAB IDE **–** User-generated code that is entered when an interrupt occurs. The location of the code in program memory will usually depend on the type of interrupt that has occurred.

**Interrupt Vector**

Address of an interrupt service routine or interrupt handler.

## L

**L-value**

An expression that refers to an object that can be examined and/or modified. An l-value expression is used on the left-hand side of an assignment.

**Latency**

The time between an event and its response.

**Library/Librarian**

*See* Archive/Archiver.

**Linker**

A language tool that combines object files and libraries to create executable code, resolving references from one module to another.

**Linker Script Files**

Linker script files are the command files of a linker. They define linker options and describe available memory on the target platform.

**Listing Directives**

Listing directives are those directives that control the assembler listing file format. They allow the specification of titles, pagination and other listing control.

**Listing File**

A listing file is an ASCII text file that shows the machine code generated for each C source statement, assembly instruction, assembler directive, or macro encountered in a source file.

**Little Endian**

A data ordering scheme for multibyte data whereby the LSB is stored at the lower addresses.

**Local Label**

A local label is one that is defined inside a macro with the LOCAL directive. These labels are particular to a given instance of a macro's instantiation. In other words, the symbols and labels that are declared as local are no longer accessible after the ENDM macro is encountered.

**Logic Probes**

Up to 14 logic probes can be connected to some Microchip emulators. The logic probes provide external trace inputs, trigger output signal, +5V, and a common ground.

**Loop-Back Test Board**

Used to test the functionality of the MPLAB REAL ICE in-circuit emulator.

**LVDS**

Low Voltage Differential Signaling. A low noise, low-power, low amplitude method for high-speed (gigabits per second) data transmission over copper wire.

With standard I/O signaling, data storage is contingent upon the actual voltage level. Voltage level can be affected by wire length (longer wires increase resistance, which lowers voltage). But with LVDS, data storage is distinguished only by positive and negative voltage values, not the voltage level. Therefore, data can travel over greater lengths of wire while maintaining a clear and consistent data stream.

Source: http://www.webopedia.com/TERM/L/LVDS.html.

## M

**Machine Code**

The representation of a computer program that is actually read and interpreted by the processor. A program in binary machine code consists of a sequence of machine instructions (possibly interspersed with data). The collection of all possible instructions for a particular processor is known as its "instruction set."

**Machine Language**

A set of instructions for a specific central processing unit, designed to be usable by a processor without being translated.

**Macro**

Macro instruction. An instruction that represents a sequence of instructions in abbreviated form.

**Macro Directives**

Directives that control the execution and data allocation within macro body definitions.

**Makefile**

Export to a file the instructions to Make the project. Use this file to Make your project outside of MPLAB IDE, i.e., with a `make`.

**Make Project**

A command that rebuilds an application, recompiling only those source files that have changed since the last complete compilation.

**MCU**

Microcontroller Unit. An abbreviation for microcontroller. Also uC.

**Memory Model**

For C compilers, a representation of the memory available to the application. For the PIC18 C compiler, a description that specifies the size of pointers that point to program memory.

**Message**

Text displayed to alert you to potential problems in language tool operation. A message will not stop operation.

**Microcontroller**

A highly integrated chip that contains a CPU, RAM, program memory, I/O ports and timers.

**Microcontroller Mode**

One of the possible program memory configurations of PIC18 microcontrollers. In microcontroller mode, only internal execution is allowed. Thus, only the on-chip program memory is available in microcontroller mode.

**Microprocessor Mode**

One of the possible program memory configurations of PIC18 microcontrollers. In microprocessor mode, the on-chip program memory is not used. The entire program memory is mapped externally.

**Mnemonics**

Text instructions that can be translated directly into machine code. Also referred to as opcodes.

**Module**

The preprocessed output of a source file after preprocessor directives have been executed. Also known as a translation unit.

**MPASM™ Assembler**

Microchip Technology's relocatable macro assembler for PIC microcontroller devices, KeeLoq® devices and Microchip memory devices.

**MPLAB *Language Tool* for *Device***

Microchip's C compilers, assemblers and linkers for specified devices. Select the type of language tool based on the device you will be using for your application, e.g., if you will be creating C code on a PIC18 MCU, select the MPLAB C Compiler for PIC18 MCUs.

**MPLAB ICD**

Microchip's in-circuit debuggers that works with MPLAB IDE. *See* ICE/ICD.

**MPLAB IDE**

Microchip's Integrated Development Environment. MPLAB IDE comes with an editor, project manager and simulator.

**MPLAB PM3**

A device programmer from Microchip. Programs PIC18 microcontrollers and dsPIC digital signal controllers. Can be used with MPLAB IDE or stand-alone. Replaces PRO MATE II.

**MPLAB REAL ICE™ In-Circuit Emulator**

Microchip's next-generation in-circuit emulators that works with MPLAB IDE. *See* ICE/ICD.

**MPLAB SIM**

Microchip's simulator that works with MPLAB IDE in support of PIC MCU and dsPIC DSC devices.

**MPLIB™ Object Librarian**

Microchip's librarian that can work with MPLAB IDE. MPLIB librarian is an object librarian for use with COFF object modules created using either MPASM assembler (mpasm or mpasmwin v2.0) or MPLAB C18 C compiler.

**MPLINK™ Object Linker**

MPLINK linker is an object linker for the Microchip MPASM assembler and the Microchip C18 C compiler. MPLINK linker also can be used with the Microchip MPLIB librarian. MPLINK linker is designed to be used with MPLAB IDE, though not required.

**MRU**

Most Recently Used. Refers to files and windows available to be selected from MPLAB IDE main pull down menus.

## N

**Native Data Size**

For Native trace, the size of the variable used in a Watch window must be of the same size as the selected device's data memory: bytes for PIC18 devices and words for 16-bit devices.

**Nesting Depth**

The maximum level to which macros can include other macros.

**Node**

MPLAB IDE project component.

**Non-Extended Mode (PIC18 MCUs)**

In Non-Extended mode, the compiler will not utilize the extended instructions nor the indexed with literal offset addressing.

**Non Real Time**

Refers to the processor at a breakpoint or executing single-step instructions or MPLAB IDE being run in simulator mode.

**Non-Volatile Storage**

A storage device whose contents are preserved when its power is off.

**NOP**

No Operation. An instruction that has no effect when executed except to advance the program counter.

## O

**Object Code/Object File**

Object code is the machine code generated by an assembler or compiler. An object file is a file containing machine code and, possibly, debug information. It can be immediately executable or it can be relocatable, requiring linking with other object files, e.g., libraries, to produce a complete executable program.

**Object File Directives**

Directives that are used only when creating an object file.

**Octal**

The base 8 number system that only uses the digits 0-7. The rightmost digit counts ones, the next digit counts multiples of 8, then $8^2$ = 64, etc.

**Off-Chip Memory**

Off-chip memory refers to the memory selection option for the PIC18 device where memory can reside on the target board, or where all program memory can be supplied by the emulator. The **Memory** tab accessed from *Options>Development Mode* provides the Off-Chip Memory selection dialog box.

**Opcodes**

Operational Codes. *See* Mnemonics.

**Operators**

Symbols, like the plus sign '+' and the minus sign '-', that are used when forming well-defined expressions. Each operator has an assigned precedence that is used to determine order of evaluation.

**OTP**

One Time Programmable. EPROM devices that are not in windowed packages. Since EPROM needs ultraviolet light to erase its memory, only windowed devices are erasable.

## P

**Pass Counter**

A counter that decrements each time an event (such as the execution of an instruction at a particular address) occurs. When the pass count value reaches zero, the event is satisfied. You can assign the Pass Counter to break and trace logic, as well as any sequential event in the complex trigger dialog.

**PC**

Personal Computer or Program Counter.

**PC Host**

Any PC running a supported Windows operating system.

**Persistent Data**

Data that is never cleared or initialized. Its intended use is so that an application can preserve data across a device Reset.

**Phantom Byte**

An unimplemented byte in the dsPIC architecture that is used when treating the 24-bit instruction word as if it were a 32-bit instruction word. Phantom bytes appear in dsPIC hex files.

**PIC MCUs**

PIC microcontrollers (MCUs) refers to all Microchip microcontroller families.

**PICkit 2 and 3**

Microchip's developmental device programmers with debug capability through Debug Express. See the Readme files for each tool to see which devices are supported.

**Plug-ins**

The MPLAB IDE has both built-in components and plug-in modules to configure the system for a variety of software and hardware tools. Several plug-in tools can be found under the Tools menu.

**Pod**

The enclosure for an in-circuit emulator or debugger. Other names are "Puck", if the enclosure is round, and "Probe", not be confused with logic probes.

**Power-on-Reset Emulation**

A software randomization process that writes random values in data RAM areas to simulate uninitialized values in RAM upon initial power application.

**Pragma**

A directive that has meaning to a specific compiler. Often a pragma is used to convey implementation-defined information to the compiler. MPLAB C30 uses attributes to convey this information.

**Precedence**

Rules that define the order of evaluation in expressions.

**Production Programmer**

A production programmer is a programming tool that has resources designed in to program devices rapidly. It has the capability to program at various voltage levels and completely adheres to the programming specification. Programming a device as fast as possible is of prime importance in a production environment where time is of the essence as the application circuit moves through the assembly line.

**Profile**

For MPLAB SIM simulator, a summary listing of executed stimulus by register.

**Program Counter**

The location that contains the address of the instruction that is currently executing.

**Program Counter Unit**

16-bit assembler – A conceptual representation of the layout of program memory. The program counter increments by 2 for each instruction word. In an executable section, 2 program counter units are equivalent to 3 bytes. In a read-only section, 2 program counter units are equivalent to 2 bytes.

**Program Memory**

MPLAB IDE – The memory area in a device where instructions are stored. Also, the memory in the emulator or simulator containing the downloaded target application firmware.

16-bit assembler/compiler – The memory area in a device where instructions are stored.

**Project**

A project contains the files needed to build an application (source code, linker script files, etc.) along with their associations to various build tools and build options.

**Prologue**

A portion of compiler-generated code that is responsible for allocating stack space, preserving registers and performing any other machine-specific requirement specified in the runtime model. This code executes before any user code for a given function.

**Prototype System**

A term referring to a user's target application, or target board.

**Psect**

The OCG equivalent of a GCC section, short for program section. A block of code or data which is treated as a whole by the linker.

**PWM Signals**

Pulse Width Modulation Signals. Certain PIC MCU devices have a PWM peripheral.

## Q

**Qualifier**

An address or an address range used by the Pass Counter or as an event before another operation in a complex trigger.

## R

**Radix**

The number base, hex, or decimal, used in specifying an address.

**RAM**

Random Access Memory (Data Memory). Memory in which information can be accessed in any order.

**Raw Data**

The binary representation of code or data associated with a section.

**Read Only Memory**

Memory hardware that allows fast access to permanently stored data but prevents addition to or modification of the data.

**Real Time**

When an in-circuit emulator or debugger is released from the halt state, the processor runs in Real Time mode and behaves exactly as the normal chip would behave. In Real Time mode, the real time trace buffer of an emulator is enabled and constantly captures all selected cycles, and all break logic is enabled. In an in-circuit emulator or debugger, the processor executes in real time until a valid breakpoint causes a halt, or until the user halts the execution.

In the simulator, real time simply means execution of the microcontroller instructions as fast as they can be simulated by the host CPU.

**Recursive Calls**

A function that calls itself, either directly or indirectly.

**Recursion**

The concept that a function or macro, having been defined, can call itself. Great care should be taken when writing recursive macros; it is easy to get caught in an infinite loop where there will be no exit from the recursion.

**Reentrant**

A function that can have multiple, simultaneously active instances. This can happen due to either direct or indirect recursion or through execution during interrupt processing.

**Relaxation**

The process of converting an instruction to an identical, but smaller instruction. This is useful for saving on code size. MPLAB ASM30 currently knows how to RELAX a CALL instruction into an RCALL instruction. This is done when the symbol that is being called is within +/- 32k instruction words from the current instruction.

**Relocatable**

An object whose address has not been assigned to a fixed location in memory.

**Relocatable Section**

16-bit assembler – A section whose address is not fixed (absolute). The linker assigns addresses to relocatable sections through a process called relocation.

**Relocation**

A process performed by the linker in which absolute addresses are assigned to relocatable sections and all symbols in the relocatable sections are updated to their new addresses.

**ROM**

Read Only Memory (Program Memory). Memory that cannot be modified.

**Run**

The command that releases the emulator from halt, allowing it to run the application code and change or respond to I/O in real time.

**Run-time Model**

Describes the use of target architecture resources.

**Runtime Watch**

A Watch window where the variables change in as the application is run. See individual tool documentation to determine how to set up a runtime watch. Not all tools support runtime watches.

## S

**Scenario**

For MPLAB SIM simulator, a particular setup for stimulus control.

**Section**

The GCC equivalent of an OCG psect. A block of code or data which is treated as a whole by the linker.

**Section Attribute**

A GCC characteristic ascribed to a section (e.g., an `access` section).

**Sequenced Breakpoints**

Breakpoints that occur in a sequence. Sequence execution of breakpoints is bottom-up; the last breakpoint in the sequence occurs first.

**Serialized Quick Turn Programming**

Serialization allows you to program a serial number into each microcontroller device that the Device Programmer programs. This number can be used as an entry code, password or ID number.

**Shell**

The MPASM assembler shell is a prompted input interface to the macro assembler. There are two MPASM assembler shells: one for the DOS version and one for the Windows version.

**Simulator**

A software program that models the operation of devices.

**Single Step**

This command steps though code, one instruction at a time. After each instruction, MPLAB IDE updates register windows, watch variables, and status displays so you can analyze and debug instruction execution. You can also single step C compiler source code, but instead of executing single instructions, MPLAB IDE will execute all assembly level instructions generated by the line of the high level C statement.

**Skew**

The information associated with the execution of an instruction appears on the processor bus at different times. For example, the executed opcodes appears on the bus as a fetch during the execution of the previous instruction, the source data address and value and the destination data address appear when the opcodes is actually executed, and the destination data value appears when the next instruction is executed. The trace buffer captures the information that is on the bus at one instance. Therefore, one trace buffer entry will contain execution information for three instructions. The number of captured cycles from one piece of information to another for a single instruction execution is referred to as the skew.

**Skid**

When a hardware breakpoint is used to halt the processor, one or more additional instructions can be executed before the processor halts. The number of extra instructions executed after the intended breakpoint is referred to as the skid.

**Source Code**

The form in which a computer program is written by the programmer. Source code is written in a formal programming language which can be translated into machine code or executed by an interpreter.

**Source File**

An ASCII text file containing source code.

**Special Function Registers (SFRs)**

The portion of data memory (RAM) dedicated to registers that control I/O processor functions, I/O status, timers or other modes or peripherals.

**SQTP**

*See* Serialized Quick Turn Programming.

**Stack, Hardware**

Locations in PIC microcontroller where the return address is stored when a function call is made.

**Stack, Software**

Memory used by an application for storing return addresses, function parameters, and local variables. This memory is dynamically allocated at runtime by instructions in the program. It allows for reentrant function calls.

**Stack, Compiled**

A region of memory managed and allocated by the compiler in which variables are statically assigned space. It replaces a software stack when such mechanisms cannot be efficiently implemented on the target device. It precludes reentrancy.

**MPLAB Starter Kit for *Device***

Microchip's starter kits contains everything needed to begin exploring the specified device. View a working application and then debug and program you own changes.

**Static RAM or SRAM**

Static Random Access Memory. Program memory you can read/write on the target board that does not need refreshing frequently.

**Status Bar**

The Status Bar is located on the bottom of the MPLAB IDE window and indicates such current information as cursor position, development mode and device, and active tool bar.

**Step Into**

This command is the same as Single Step. Step Into (as opposed to Step Over) follows a CALL instruction into a subroutine.

**Step Over**

Step Over allows you to debug code without stepping into subroutines. When stepping over a `CALL` instruction, the next breakpoint will be set at the instruction after the `CALL`. If for some reason the subroutine gets into an endless loop or does not return properly, the next breakpoint will never be reached. The Step Over command is the same as Single Step except for its handling of `CALL` instructions.

**Step Out**

Step Out allows you to step out of a subroutine which you are currently stepping through. This command executes the rest of the code in the subroutine and then stops execution at the return address to the subroutine.

**Stimulus**

Input to the simulator, i.e., data generated to exercise the response of simulation to external signals. Often the data is put into the form of a list of actions in a text file. Stimulus can be asynchronous, synchronous (pin), clocked and register.

**Stopwatch**

A counter for measuring execution cycles.

**Storage Class**

Determines the lifetime of the memory associated with the identified object.

**Storage Qualifier**

Indicates special properties of the objects being declared (e.g., `const`).

**Symbol**

A symbol is a general purpose mechanism for describing the various pieces which comprise a program. These pieces include function names, variable names, section names, file names, struct/enum/union tag names, etc. Symbols in MPLAB IDE refer mainly to variable names, function names and assembly labels. The value of a symbol after linking is its value in memory.

**Symbol, Absolute**

Represents an immediate value such as a definition through the assembly `.equ` directive.

**System Window Control**

The system window control is located in the upper left corner of windows and some dialogs. Clicking on this control usually pops up a menu that has the items “Minimize,” “Maximize,” and “Close.”

## T

**Target**

Refers to user hardware.

**Target Application**

Software residing on the target board.

**Target Board**

The circuitry and programmable device that makes up the target application.

**Target Processor**

The microcontroller device on the target application board.

**Template**

Lines of text that you build for inserting into your files at a later time. The MPLAB Editor stores templates in template files.

**Tool Bar**

A row or column of icons that you can click on to execute MPLAB IDE functions.

**Trace**

An emulator or simulator function that logs program execution. The emulator logs program execution into its trace buffer which is uploaded to MPLAB IDE's trace window.

**Trace Memory**

Trace memory contained within the emulator. Trace memory is sometimes called the trace buffer.

**Trace Macro**

A macro that will provide trace information from emulator data. Since this is a software trace, the macro must be added to code, the code must be recompiled or reassembled, and the target device must be programmed with this code before trace will work.

**Trigger Output**

Trigger output refers to an emulator output signal that can be generated at any address or address range, and is independent of the trace and breakpoint settings. Any number of trigger output points can be set.

**Trigraphs**

Three-character sequences, all starting with ?? (two consecutive question marks), that are defined by ISO C as replacements for single characters.

## U

**Unassigned Section**

A section which has not been assigned to a specific target memory block in the linker command file. The linker must find a target memory block in which to allocate an unassigned section.

**Uninitialized Data**

Data which is defined without an initial value. In C,

```
int myVar;
```

defines a variable which will reside in an uninitialized data section.

**Upload**

The Upload function transfers data from a tool, such as an emulator or programmer, to the host PC or from the target board to the emulator.

**USB**

Universal Serial Bus. An external peripheral interface standard for communication between a computer and external peripherals over a cable using bi-serial transmission. USB 1.0/1.1 supports data transfer rates of 12 Mbps. Also referred to as high-speed USB, USB 2.0 supports data rates up to 480 Mbps.

## V

**Vector**

The memory locations that an application will jump to when either a Reset or interrupt occurs.

**Volatile**

A variable qualifier which prevents the compiler applying optimizations that affect how the variable is accessed in memory.

## W

**Warning**

MPLAB IDE – An alert that is provided to warn you of a situation that would cause physical damage to a device, software file, or equipment.

16-bit assembler/compiler – Warnings report conditions that can indicate a problem, but do not halt processing. In MPLAB C30, warning messages report the source file name and line number, but include the text '`warning`:' to distinguish them from error messages.

**Watch Variable**

A variable that you can monitor during a debugging session in a Watch window.

**Watch Window**

Watch windows contain a list of watch variables that are updated at each breakpoint.

**Watchdog Timer (WDT)**

A timer on a PIC microcontroller that resets the processor after a selectable length of time. The WDT is enabled or disabled and set up using Configuration bits.

**Workbook**

For MPLAB SIM stimulator, a setup for generation of SCL stimulus.

**NOTES:**

# Index

# Worldwide Sales and Service

## AMERICAS

**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:
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

## ASIA/PACIFIC

**Asia Pacific Office**
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon

**Hong Kong**
Tel: 852-2943-5100
Fax: 852-2401-3431

**Australia - Sydney**
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

**China - Beijing**
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

**China - Chengdu**
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

**China - Chongqing**
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

**China - Dongguan**
Tel: 86-769-8702-9880

**China - Guangzhou**
Tel: 86-20-8755-8029

**China - Hangzhou**
Tel: 86-571-8792-8115
Fax: 86-571-8792-8116

**China - Hong Kong SAR**
Tel: 852-2943-5100
Fax: 852-2401-3431

**China - Nanjing**
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

**China - Qingdao**
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

**China - Shanghai**
Tel: 86-21-3326-8000
Fax: 86-21-3326-8021

**China - Shenyang**
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

**China - Shenzhen**
Tel: 86-755-8864-2200
Fax: 86-755-8203-1760

**China - Wuhan**
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

**China - Xian**
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

## ASIA/PACIFIC

**China - Xiamen**
Tel: 86-592-2388138
Fax: 86-592-2388130

**China - Zhuhai**
Tel: 86-756-3210040
Fax: 86-756-3210049

**India - Bangalore**
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

**India - New Delhi**
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

**India - Pune**
Tel: 91-20-3019-1500

**Japan - Osaka**
Tel: 81-6-6152-7160
Fax: 81-6-6152-9310

**Japan - Tokyo**
Tel: 81-3-6880- 3770
Fax: 81-3-6880-3771

**Korea - Daegu**
Tel: 82-53-744-4301
Fax: 82-53-744-4302

**Korea - Seoul**
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

**Malaysia - Kuala Lumpur**
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

**Malaysia - Penang**
Tel: 60-4-227-8870
Fax: 60-4-227-4068

**Philippines - Manila**
Tel: 63-2-634-9065
Fax: 63-2-634-9069

**Singapore**
Tel: 65-6334-8870
Fax: 65-6334-8850

**Taiwan - Hsin Chu**
Tel: 886-3-5778-366
Fax: 886-3-5770-955

**Taiwan - Kaohsiung**
Tel: 886-7-213-7830

**Taiwan - Taipei**
Tel: 886-2-2508-8600
Fax: 886-2-2508-0102

**Thailand - Bangkok**
Tel: 66-2-694-1351
Fax: 66-2-694-1350

## EUROPE

**Austria - Wels**
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

**Denmark - Copenhagen**
Tel: 45-4450-2828
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

**France - Saint Cloud**
Tel: 33-1-30-60-70-00

**Germany - Garching**
Tel: 49-8931-9700

**Germany - Haan**
Tel: 49-2129-3766400

**Germany - Heilbronn**
Tel: 49-7131-67-3636

**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: 47-7289-7561

**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