AVR® GNU Toolchain to MPLAB® XC8 Migration Guide

Notice to Development Tools Customers

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

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

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

Table of Contents

1. Preface

1.1 Conventions Used in This Guide

The following conventions may appear in this documentation:

Table 1-1. Documentation Conventions

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	*AVR® GNU Toolchain to MPLAB® XC8 Migration Guide*
	Emphasized text	...is the *only* compiler...
Initial caps	A window	the Output window
	A dialog	the Settings dialog
	A menu selection	Select File and then Save.
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:\Users\User1\Projects`
	Keywords	`static`, `auto`, `extern`
	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	`xc8 [`*`options`*`] `*`files`*
Curly brackets and pipe character: { | }	Choice of mutually exclusive arguments; an OR selection	`errorlevel {0|1}`

.....continued		
Description	**Represents**	**Examples**
Ellipses...	Replaces repeated text	*var_name* [, *var_name*...]
	Represents code supplied by user	void main (void) { ... }

1.2 Recommended Reading

This guide describes the changes to source code and build options that might be required should you decide to migrate a C-based project from the AVR® 8-bit GNU Toolchain to the Microchip MPLAB® XC8 C Compiler. The following Microchip documents are also available and are recommended as supplemental reference resources.

MPLAB® XC8 C Compiler User's Guide for AVR® MCU
This version of the compiler's user's guide is for projects that target 8-bit AVR devices.

MPLAB® XC8 C Compiler Release Notes for AVR® MCU
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 the latest information and known issues that might not be included in the C Compiler's user's guide.

Microchip Unified Standard Library Reference Guide
This guide contains information and examples relating to the types, macros, and functions defined by the C standard libraries and which is shipped with all MPLAB XC C compilers.

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

2. Migration Overview

This guide describes the changes to source code and build options that might be required should you decide to migrate a C-based project from the AVR® 8-bit GNU Toolchain (referred to as AVR-GCC in this guide) to the Microchip MPLAB® XC8 C Compiler (MPLAB XC8).

Detailed in this guide are issues that must be considered to ensure that the migrated project works as expected. There should be little difference in implementation-defined behavior between the two compilers, but prudent programmers will not write code whose operation is dependent on such behavior.

In addition to the standard GCC feature set provided by AVR-GCC, MPLAB XC8 implements many unique language extensions and features. Some of these are also presented in this guide and should be explored to take full advantage of the tool. Many of these features are part of the Common C Interface (CCI). Code that complies with this interface can be more easily ported across all MPLAB XC compilers.

A migrated project will produce a HEX file that differs to that built using the original project and the AVR-GCC compiler, so any hash values calculated from the final program image will need to be recalculated. Objects and functions will almost certainly be linked at different addresses. Differing code generation strategies and optimizations might affect any code that relies on the timing of its execution.

See the *MPLAB® XC8 C Compiler User's Guide for AVR™ MCU* for full information on how to use the MPLAB XC8 compiler and for more detailed information on the compiler's extensions and features.

2.1 Introduction to the 8-bit AVR Toolchains

Microchip distributes two toolchains that target 8-bit AVR devices, these being the AVR® 8-bit GNU Toolchain (AVR-GCC) and the MPLAB® XC8 C Compiler (MPLAB XC8). When targeting AVR devices, both toolchains use applications based on the open-source GCC C/C++ compiler developed by GNU (www.gnu.org). Both compilers target all 8-bit AVR devices, although the MPLAB XC8 compiler additionally targets all 8-bit PIC devices. The compilers differ in their features that support ease of coding and optimization, as well as their requirement of a purchased license to operate with full optimization, as summarized in the following table.

Table 2-1. Comparison of AVR-GCC and MPLAB XC8 Features

Compiler (License)	Cost	Supported devices	Coding features
AVR-GCC (Unlicensed)	Free	8-bit AVR	Basic
MPLAB XC8 (Unlicensed)	Free	8-bit AVR and PIC	Intermediate
MPLAB XC8 (Licensed)	Purchased	8-bit AVR and PIC	Advanced

3. Differences Requiring Attention

This section deals with potential issues that must be considered when migrating code from AVR-GCC to MPLAB XC8. Failure to correctly address these might result in code failure.

3.1 Const-in-program-memory Feature

For some devices, the MPLAB XC8 compiler can employ a const-in-progmem feature that automatically places any `const`-qualified objects with static storage duration into program memory rather than into RAM. When enabled, this feature affects where such objects are placed, how they are accessed, and the size and format of pointers that might indirectly access them.

Important: For some devices, this feature is enabled by default if no controlling option is specified, and it can adversely affect program operation.

Description

The 8-bit AVR devices fall into three groups with regard to how program Flash memory is mapped, those being:

1. Devices that map their entire Flash memory into the data space (such as those in the tinyAVR® or AVR XMEGA® 3 families),
2. Devices with configurable Flash mapping, which can map part of their Flash memory into the data space (such as those in the AVRDA family), and
3. Devices that do not map any part of Flash memory into the data space.

Check the data sheet for your device to see if and how Flash memory is mapped into the data space. If you are not sure, the preprocessor macro `__AVR_PROGMEM_IN_DATA_ADDRESS_SPACE__` is defined by the compiler for devices in group (1) or (2), provided the `-mno-const-data-in-progmem` has not been used. The macro `__AVR_CONST_DATA_IN_CONFIG_MAPPED_PROGMEM__` is defined for devices in group (2), provided the `-mno-const-data-in-config-mapped-progmem` option has not been used. To see macros defined by the compiler, use the `-E` and `-dM` options with your project and check the definitions printed in the standard output stream.

Group (1) Devices

The const-in-progmem feature is not required and not used with any device from group (1), which map their program memory into the data space. Programs built for such devices have `const`-qualified objects with static storage duration placed into the mapped Flash region where they can be read from data memory. For these devices, this compiler feature is not necessary, and the placement and access of `const`-qualified objects is similar in both compilers.

Group (2) Devices

When building for devices in group (2), which map a block of program memory into the data space, the const-in-config-mapped-progmem feature can be used to control where `const`-qualified objects are placed and accessed. This feature is enabled by the `-mconst-data-in-config-mapped-progmem` option, or by default if no option is specified. When enabled, devices in group (2) are treated similarly to devices in group (1). All `const`-qualified objects with static storage duration are placed into a Flash region that is mapped by the runtime startup code, and which can be read from data memory. The size of this mapped Flash region, however, is limited to a single 32 KB block that is chosen by the linker.

This behavior can be disabled using the `-mno-const-data-in-config-mapped-progmem` option, so that `const`-qualified objects are stored in and read from non-mapped program memory, and the device is then treated similarly to those in group (3). And as is the case for other devices in group (3), the const-in-progmem feature can be disabled entirely using the `-mno-const-data-in-progmem` option, forcing the compiler to copy `const`-qualified data to RAM, where it can be read. In this case, the behavior is compatible with AVR GCC.

Group (3) Devices

When building for devices in group (3), which do not map program memory in the data space, the const-in-progmem feature can be used to control where `const`-qualified objects are placed and accessed. This feature is enabled by the `-mconst-data-in-progmem` option, or by default if no option is specified. It can be disabled by using the `-mno-const-data-in-progmem` option.

When the feature is **disabled**, the operation of MPLAB XC8 is largely compatible with that of AVR-GCC. Initial values for `const`-qualified objects are placed in program memory but are copied to a data memory location by the runtime startup code where they are accessed at runtime. The same is true for string literals. Additionally, data pointers not using any non-standard specifiers will indirectly access RAM objects, with any use of the `const` qualifier indicating only that the target is read-only. To be able to read program memory indirectly, the pointer must use a non-standard specifier, such as `__memx` or `__flashn`. For example:

```
const char * cp1;         // I can read objects in RAM
char * cp2;               // I can read and write objects in RAM
const __memx char * cp3;  // I can read objects in RAM or program memory
```

One consequence of this is that any function that takes a pointer argument (either `const` or non-`const` qualified target), for example, can only be passed the address of objects in the data memory space. This is particularly relevant for functions in the standard C library. An alternate function must be supplied and called if the action has to be performed using the address of an object in program memory.

When the const-in-progmem feature is **enabled** (the default action if no option has been specified), `const`-qualified objects are instead placed in and read from program memory. String literals are handled similarly. Such objects are read using alternate instruction sequences that are typically longer than the equivalent sequences that read from data memory, but less data memory is used by the program.

In addition, pointers to `const`-qualified types can indirectly read objects in either data or program memory, and a bit encoded into addresses assigned to such pointers determines which space is to be accessed. Essentially, these pointers act like their type had been specified with `__memx` in addition to the `const` qualifier. Code to dereference the pointer is larger, but because they can access either memory space, no duplication of pointers or functions is required to handle objects in each space.

Migration

When migrating code to MPLAB XC8, there are two alternatives for code running on devices considered as being in group (3) (including devices from group (2) that have disabled the mapping feature):

- Disable the const-in-progmem feature and ensure legacy library routines that read from program memory are linked in with the project code.
- Leave the const-in-progmem feature enabled and ensure that the project source code is not making incorrect assumptions about the location of `const`-qualified objects.

These two migration strategies are discussed below.

Disable the const-in-progmem Feature

To follow the first migration approach, disable this feature by deselecting the "Enables access of const variables directly from flash" checkbox in the “XC8 Global option” category in the MPLAB X IDE Project Properties dialog. Disabling this feature will ensure that `const`-qualified objects with static storage duration are copied to and accessed from data memory. Any pointer to a `const`-qualified type will read only from data memory.

AVR-GCC provides program-memory alternatives to some standard C library routines that take pointer arguments. These alternatives use a `_P` suffix and are written so that they will read from program memory, for example `strlen_P()`, specified in `<avr/pgmspace.h>`, which can be called to find the length of a string in program memory. These functions are not supplied with MPLAB XC8; however, you can download the AVR-libc to obtain the source files for these functions.

This Online Microchip Help web page documents the AVR-libc. It also contains a link to the AVR Libc web page where you can find a link to the download area. Download the required AVR-libc archive and unzip it to a local directory. You can include source files into your projects directly from this local directory or copy the required source files into your project directory and then add them to your project from that location. Note that some functions are written in assembly code.

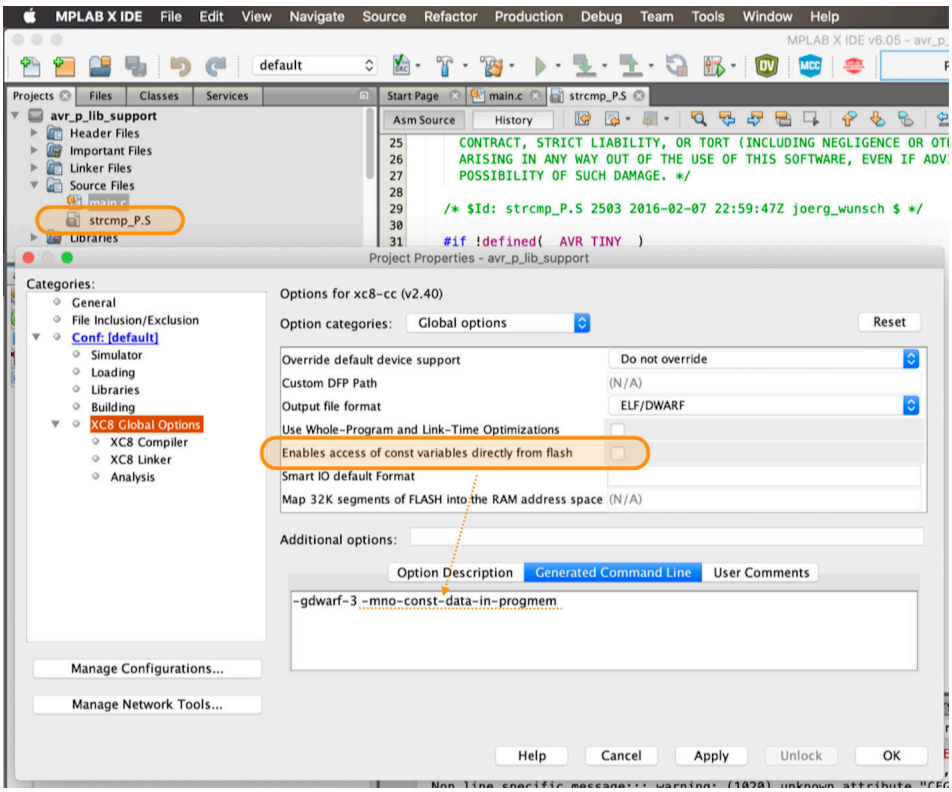
Your program will need declarations from the `<avr/pgmspace.h>` header, which can be found in the downloaded library. Note that this header itself also includes other header files. You can provide the MPLAB XC8 preprocessor

with the path to this header in the unzipped local directory, or alternatively the path to copies of this header in your project directory. The `-I` compiler option allows the path to be specified, and this can be specified in the “Include directories” field, found in the “Preprocessing and messages” options within the “XC8 Compiler” category of the Project Properties dialog. AVR-libc headers should be scanned first before the standard headers; however, this will be the case if you add the search path as described above.

On rare occasions, there might be conflicting information in code or headers provided by AVR-libc and those provided by the libraries shipped with MPLAB XC8. If the compiler sees a declaration from an AVR-libc header but links in the corresponding definition from the compiler-shipped library (or vice versa), the project might fail to build or execute correctly if there is a mismatch in that information. Where there is such a conflict, you could selectively copy those declarations required from AVR-libc headers and create your own header file from these.

For example, a programmer has decided to disable the const-in-progmem feature when migrating an AVR-GCC project. The "Enables access of const variables directly from flash" checkbox was unchecked in the Project Properties dialog (as shown in the figure below).

Figure 3-1. A project setup to build the program memory version of the string compare function (`strcmp_P()`) source file and disable the const-in-progmem feature (highlighted)



The original program performed a string compare between a RAM and program memory pointer, as shown below. As the `strcmp_P()` function call cannot be simply swapped to a call to `strcmp()` in the default standard C library, the source file (`strcmp_P.S`) for the `strcmp_P` function was copied from the ARV-libc archive into the project directory and added to the project's Source files folder in the MPLAB X IDE (as shown in the above figure). The source code contained in this file includes the header `<macros.inc>` (which itself includes `<sectionname.h>`). These headers were also copied to a subdirectory in the project directory and the path to that directory specified in the “Include directories” field of the Project Properties dialog. The prototype for the `strcmp_P()` function can be found in the `<avr/pgmspace.h>` header. This header was also copied to the subdirectory.

```
/* const-in-progmem Feature Disabled */

#include <string.h>
#include <pgmspace.h>

const char ro_ram_1[] = "another";          // I am in RAM
const char ro_ram_2[] = "string";           // So am I
```

```
const PROGMEM char ro_prog_1[] = "another";  // I am in program memory

volatile int diffs;

int main(void) {
    if (strcmp(ro_ram_1, ro_ram_2))          // compare RAM to RAM
        diffs++;
    if (strcmp_P(ro_ram_1, ro_prog_1))       // compare RAM to prog memory
        diffs++;
    while (1) {
    }
}
```

Leave the const-in-progmem Feature Enabled

To follow the second migration approach, ensure the const-in-progmem feature is enabled by selecting the "Enables access of const variables directly from flash" checkbox in the “XC8 Global option” category in the MPLAB X IDE Project Properties dialog. With this feature enabled, objects defined using the `const` qualifier and that have static storage duration will be placed in program memory and any pointer with a `const`-qualified target will be able to read from either data or program memory space.

Migrated projects using this feature typically fail when pointers to non-`const` objects are assigned addresses of objects in program memory. Code or linker options that expect any `const`-qualified object to be present in RAM will fail. Code that assumes a pointer has a certain address might fail. Direct access of `const` objects will work as expected.

The code shown below will not compile when this feature is enabled, due to a mismatching in types between the prototype for `readChar()` and the arguments of the last two calls.

```
char readChar(char * cp) {
  return * cp;
}

char foobar_in_ram = 0x33;
const char foobar_as_const[4] = { 3, 2, 4, 6 };

int main(void) {
  volatile char result;

  result = readChar(&foobar_in_ram);
  result = readChar(&foobar_as_const[2]);  // this won't work
  result = readChar("hi there");           // neither will this
}
```

The temptation is to cast the address of these calls so they match the prototype of the function, as follows:

```
  result = readChar((char *)&foobar_as_const[2]);     // No!
  result = readChar((char *)"hi there");              // No!
```

Such a change will indeed suppress the errors; however the code will fail when executing. If a pointer needs to be able to access program memory, ensure it is a pointer to a `const`-qualified type, that is, the prototype for `readChar()` needs to be `char readChar(const char * cp)`.

The use of separate routines to access different memory spaces is no longer required. Always use the standard C string functions when the const-in-progmem feature is enabled. For example, you do not need to choose between calling `strcmp()` or `strcmp_P()` to compare strings; the standard `strcmp()` function provided in the standard library should be used to compare any string, regardless of where it is located.

```
/* const-in-progmem Feature Enabled */

#include <string.h>

char ro_ram_1[] = "another";                 // I am in RAM
char ro_ram_2[] = "string";                  // So am I
const PROGMEM char ro_prog_1[] = "another";  // I am in program memory

volatile int diffs;

int main(void) {
    if (strcmp(ro_ram_1, ro_ram_2))          // compare RAM to RAM
        diffs++;
```

```
    if (strcmp(ro_ram_1, ro_prog_1))         // compare RAM to prog memory
        diffs++;
    while (1) {
    }
}
```

The use of the non-standard (upper-case) `%S` printf-family conversion specifier provided by AVR-libc is also not required when the const-in-progmem feature is enabled. Use the standard `%s` specifier to print any string, for example:

```
    printf("These are the RAM characters: %s\n", ro_ram_1);
    printf("These are the ROM characters: %s\n", ro_prog_1);
```

3.2 Link Order

When using MPLAB XC8, the order in which the linker will process object files can differ to that used by AVR-GCC. This might affect how sections within those files are positioned in memory.

Description

Whether building a project using an IDE or make files on the command-line, you can specify the order in which source files are presented to the MPLAB XC8 compiler driver. The driver then passes to the linker application the object files generated from these source files, any object or library archives specified in the project or on the command line, and any object or library archives automatically linked in by the compiler driver. This operation is similar for both compilers; however, even though files are passed to each linker in the same order, the way in which these files are internally processed by the linker and hence the order in which the sections within those files are allocated memory can differ.

With MPLAB XC8, sections are allocated using a best-fit allocator (BFA) when the `-mrelax` option has not been specified. When this option has been specified, it uses a section-references-based heuristic to increase the linker's chance of shortening long jumps and calls. In either case, functions and objects might be processed in different orders and hence allocated at different addresses to those allocated when building the same project with AVR-GCC.

Migration

There is no option or code feature to control the order in which the MPLAB XC8 linker processes object files, hence where the sections are allocated memory. If the process order of these files is critical to your application, you will need to create a custom linker script that explicitly lists the object files in the required order. For example, the `.text` output section could be specified as follows:

```
.text // Output section
{
   ...
   *objfile1.o(.text) *objfile1.o(.text.*) // All .text and .text.* from objfile1.o
   *objfile2.o(.text) *objfile2.o(.text.*) // All .text and .text.* from objfile2.o
   ...
}
```

When processed, `.text` and `.text.*` sections from `objfile1.o` are allocated first, followed by those from `objfile2.o`, leaving the linker to allocate however it chooses only those sections that have not been specified by the linker script.

The linker script will need to define similar output sections for the other input sections and must also specify the names of object files or library archives that are automatically linked in by the compiler driver.

3.3 Moving Data and Code Sections

Unlike the AVR GCC compiler, which typically produces a single `.text`, `.data` and `.bss` output section, this is not the case when using MPLAB XC8. Using the `--section-start` option to move code and data with MPLAB XC8 will not work reliably.

Description
One consequence of the best-fit allocator (BFA) employed by MPLAB XC8 is that most objects and code are not placed in the standard `.text`, `.data` and `.bss` output sections . The compiler only maps the vector table, `.dinit` table and some special sections into the `.text` output section, leaving other `.text*` input sections unmapped. The linker implementation then creates an output section for each unmapped input section and proceeds to allocate them as it sees fit.

If you need to explicitly locate any of these standard sections, the `-Wl,--section-start` option will not work as expected. Additionally, any tools used to dump or analyze the output after linking and that expect to see a single `.text` section for all the program code might also fail.

Provided you are using the latest MPLAB XC8 compiler, the `-Wl,--section-start` option can be used to position sections with a user-defined name (sections not called `.text`, `.data` and `.bss`) and without any modification to the linker script. User-defined sections can be created using the `__section()` specifier.

Migration
The `-Wl,-T`*`section=addr`* option can instead be used to position standard sections, like the `.text`, `.data` and `.bss` sections at the specified address.

Although the `-Wl,-Ttext` option will move the `.text` section to the offset specified, this option also sets the starting address from which the linker begins program memory allocation for all code (even that handled by the BFA). Thus, when this option is used, the `.text` section will start at the exact offset specified; other code sections will be located after this. Similarly, the `-Wl,-Tdata` option also sets the starting address from which the linker begins data (both initialized and uninitialized) allocation.

3.4 Procedural Abstraction

Procedural Abstraction optimizations, not available with the AVR-GCC compiler, are utilized by MPLAB XC8, and the effect of these might break assumptions about where code is located.

Description
Procedural Abstraction is essentially a reverse inlining process and is performed on identical assembly code sequences that appear more than once. Identical code sequences are abstracted into a callable routine, and each instance of the original code sequence is replaced with a call to that routine. This optimization reduces code size considerably, but will impose a small reduction in execution speed. This optimization can, however, adversely impact debugging.

When using MPLAB XC8, Procedural Abstraction optimizations are automatically turned on with level `s` optimizations (`-Os`). Any abstracted assembly sequences will be placed in a separate section to the original code. If there are linker options or linker scripts being used to locate code at specific addresses, these might not correctly handle the sections containing the abstracted code.

Migration
Procedural Abstraction optimizations can be turned off if they interfere with the linking requirements of a project. Completely disable Procedural Abstraction for the entire program using the `-mno-pa` option. The content of individual files or functions can be protected from Procedural Abstraction by using the `-mno-pa-on-file=`*`filename`* or `-mno-pa-on-function=`*`function`* options respectively. There is also a `nopa` attribute (or `__nopa` specifier) that can be used to disable Procedural Abstraction for a particular function.

3.5 Library Functions

The MPLAB XC8 compiler does not provide the AVR-libc standard C library for AVR-GCC. Some functions provided in AVR-libc are not implemented in the Microchip library, and projects using these functions must provide alternate definitions or be recoded.

Description

MPLAB XC8 ships with a Microchip Universal Standard Library, which is shared between all MPLAB XC compilers. This library contains only functions that form part of the C standard. Several non-standard functions are provided by AVR-libc and any project utilizing these functions will fail to build when migrated to MPLAB XC8. Some non-standard functions are only partially implemented by the Microchip Universal Standard Library and although their use will not trigger build errors, their execution will not produce the expected results.

The following table shows those functions present in AVR-lib that are not implemented or not fully implemented in the Microchip Universal Standard Library.

Functions	Header
`dtostre(),dtostrf()`	`<stdlib.h>`
`getenv()`	`<stdlib.h>`
`itoa()`	`<stdlib.h>`
`random(),srandom()`	`<stdlib.h>`
`system()`	`<stdlib.h>`
`utoa(),ultoa()`	`<stdlib.h>`
`strcasecmp(),strlwr(),strupr(), strncasecmp(),strrev()`	`<string.h>`
`ffs(),ffsl(),ffsll()`	`<string.h>`
`square()`	`<math.h>`
`mk_gmtime(),sun_rise(),sun_set(), solar_noon()`	`<time.h>`
`gmtime_r(),localtime_r()`	`<time.h>`
`asctime_r(),ctime_r(),isotime(), isotime_r()`	`<time.h>`
`set_dst(),set_zone(),set_system_time(), system_tick(),set_position()`	`<time.h>`
`is_leap_year(),month_length(), week_of_year(),month_of_year(), fatfs_time(),equation_of_time(), daylight_seconds(),moon_phase()`	`<time.h>`
`iso_week_date(),iso_week_date_r()`	`<time.h>`
`solar_declination(),gm_sideral(), lm_sideral()`	`<time.h>`

Migration

To migrate projects that have used any non-standard AVR-libc functions, the easiest option would be to download the AVR-libc to obtain the source files for these functions and include these into your project.

This Online Microchip Help web page documents the AVR-libc. It also contains a link to the AVR Libc web page where you can find a link to the download area. Download the required AVR-libc archive and unzip it to a local directory. You can include source files into your projects directly from this local directory or copy the required source files into your project directory and then add them to your project from that location. Note that some functions are written in assembly code.

The declarations for any non-standard functions can be taken from the AVR-libc headers. You can provide the MPLAB XC8 preprocessor with the path to this header in the unzipped local directory, or alternatively the path to copies of this header in your project directory. The `-I` compiler option allows the path to be specified, and this can

be specified in the “Include directories” field, found in the “Preprocessing and messages” options within the “XC8 Compiler” category of the Project Properties dialog. AVR-libc headers should be scanned first before the standard headers; however, this will be the case if you add the search path as described above.

On rare occasions, there might be conflicting information in code or headers provided by AVR-libc and those provided by the libraries shipped with MPLAB XC8. If the compiler sees a declaration from an AVR-libc header but links in the corresponding definition from the compiler-shipped library (or vice versa), the project might fail to build or execute correctly if there is a mismatch in that information. Where there is such a conflict, you could selectively copy those declarations required from AVR-libc headers and create your own header file from these.

3.6 Printf Library for Floats

The MPLAB XC8 compiler does not require nor ship the AVR-GCC library dedicated to printing float-point values. References to this library must be removed.

Description

The AVR GCC compiler ships with a floating-point-specific printf-family library (`libprintf_flt.a`). This library is not shipped with MPLAB XC8 and if the migrated project attempts to explicitly link in this library, it will fail to build. MPLAB XC8 on the other hand employs a smart-io feature to automatically link in floating-point specific code to the printing functions only when required.

Migration

No change is required to the C source code that uses any of the library functions in the printf family. If the migrated project's linker script or options refer to this library, however, these references must be removed. No replacement library is required.

The operation of the Smart IO feature is mostly automatic, however there might be instances where you can fine tune the feature's operation to reduce code size. Confirm the operation of MPLAB XC8's `-msmart-io=`*`level`* and `-msmart-io-format="`*`fmt`*`"` options in the *MPLAB® XC8 C Compiler User’s Guide for AVR™ MCU* to ensure that formatted IO in your project does not use more resource than necessary.

3.7 Custom Runtime Startup Code

Custom runtime startup code that assumes single, contiguous data or bss regions will not work as expected.

Description

In order to clear or initialize all the object and RAM function sections at runtime, the linker creates a data-initialization template that is loaded into an output section named `.dinit` and allocated in program memory. The start-up module interprets this template and writes or copies initial values into the required sections. There can be multiple non-contiguous sections present.

Migration

There is no option or code feature to directly control how MPLAB XC8 structures the runtime startup code. If control of the data initialization process is critical to your application, a linker script that maps all the `.data` and `.bss` sections would be necessary to ensure that the best-fit allocator utilized by MPLAB XC8 does not allocate them. See also 3.2. Link Order for more information on changing the linker script.

4. Things to Consider

This section deals with MPLAB XC8 source code and operational differences to AVR GCC that do not necessarily need to be migrated to have code build and execute as expected, but which when utilized properly, might reduce the complexity of your source code or result in better performance of your program.

4.1 Headers for Device-specific Resources

While MPLAB XC8 will build projects using the <avr/io.h> header, consider including <xc.h> instead.

Header files must be included into your project's source files to allow access to many device-specific resources, like special function registers (SFRs), interrupt vector names, and preprocessor macros specifying device configuration. When using AVR-GCC, the <avr/io.h> header will include the appropriate headers to define many of these resources for the selected device. All MPLAB XC projects, including those for MPLAB XC8, typically include <xc.h> as the top-level device-specific header to define the device's resources.

The <xc.h> header includes <avr/io.h>, so the definitions for SFRs and interrupt vectors will be identical, regardless of which header you include into your source files; however <xc.h> also includes other headers and defines other macros. The <xc.h> header must be included if you intend to use any Common C Interface features, described in 4.3. Common C Interface, in which case the existing inclusion of the <avr/io.h> header can be removed.

4.2 Specifying Configuration Fuses

While MPLAB XC8 will accept the code used with AVR-GCC to program the fuses in the Configuration Words, it also allows the Configuration Words to be specified in a simpler way. Consider the new constructs for new or migrated projects, but do not use a mix of programming methodologies.

MPLAB XC8 uses the config pragma to specify one or more Configuration Word setting-value pairs. The pragma ensures that the fuse data that is programmed into the device is placed in the correct section and linked at the correct address.

The general form of the pragma is:

```
#pragma config setting = value[,setting = value,...]
```

where *setting* is a fuse setting descriptor, e.g., WDTON, and *value* is a textual description of the desired state, for example CLEAR. Those states other than SET and CLEAR are prefixed with the setting descriptor, as in BODLEVEL_4V3 for the 4.3 Volt BODLEVEL setting, as shown in the following examples:

```
#pragma config SUT_CKSEL = SUT_CKSEL_14CK_62MS5
#pragma config BOOTRST = CLEAR, BOOTSZ = BOOTSZ_2048W_4800
```

For assistance with the setting-value names, open the avr_chipinfo.html file that is located in the compiler's docs directory. Click the link to your target device and the linked page will show you the pragma settings and values that are appropriate for your device, along with examples. Review your device data sheet for more information on the operation of the fuses.

4.3 Common C Interface

The Common C Interface (CCI) is available with all MPLAB XC C compilers and is designed to enhance code portability between these compilers by standardizing implementation-defined behavior and non-standard extensions. Several features available for AVR targets when using MPLAB XC8 are available only when the CCI is enabled.

The CCI can be enforced by using the -mext=cci option. There is a checkbox for this option (“Use CCI syntax”) found in the “Preprocessing and messaging” options, within the “XC8 Compiler” category in the MPLAB X IDE Project Properties for your project. When enabled, C source code is verified to ensure it is compliant with CCI. Many of the features require that the generic header <xc.h> be included; however this is typically included with MPLAB XC

compiler projects anyway, to allow access to device special function registers and resource (see 4.1. Headers for Device-specific Resources).

When the CCI is enabled, the signedness of a plain `char` type is made `unsigned`, whereas it is `signed` if no pertinent option is specified.

The following macros are available when the CCI is enabled and replace the use of similar attributes.

__align(*n*)	Align the object's address with the next n-byte boundary.
__deprecate	Generate a warning whenever the specified object is used.
__pack	Force the object or structure member to have the smallest possible alignment.
__persistent	Do not clear the object at startup.
__section(*section*)	Allocate the object to a user-nominated section.
__at(*address*)	Place the object at the specified address.
__interrupt()	Define an interrupt function.

For example, with the CCI enabled, the following code is valid:

```
#include <xc.h>

const int settings[] __at(0x260) = { 22, 5 }; //place at Flash location 0x260
int ramloc[6] __at(0x800150);                 //place at RAM location 0x150

/* SPI interrupt code */
void __interrupt(SPI_STC_vect_num) spi_Isr(void) {
  process(SPI_ClientReceive());
  return;
}
```

4.4 Linking Flash Sections

When using AVR-GCC, you must modify the linker script to ensure that Flash objects defined using the `__flash`*n* specifier appear in the correct Flash pages. This is not required when using MPLAB XC8. The linker in this toolchain automatically allocates objects to the correct address ranges. Any modification of the linker script to do with Flash object placement can be removed.

4.5 Address Units

Care needs to be exercised when interpreting and working with addresses with the MPLAB XC8 compiler and the MPLAB X IDE.

The MPLAB XC8 compiler always works with byte addresses. Addresses supplied in command-line option arguments or in code constructs like `__at(`*address*`)` must always be byte addresses. Similarly, addresses displayed in output such as map files or memory dumps will always be byte addresses. This might not be the case,however, if you are specifying an address in the MPLAB X IDE Project Properties. The IDE might map word addresses specified in Project Property fields into byte addresses to be used in the compiler build command. If in doubt, check the command line used to build the project in the Output window. Similarly, the MPLAB X IDE displays byte addresses in the Program Memory view window.

4.6 Code Size Report

The memory summary produced by AVR-GCC (and hence by the MPLAB X IDE when AVR-GCC is selected as the compiler tool) shows a lower-than-actual Flash usage. In particular, it does not consider the `.rodata` section (on devices with memory-mapped flash) as well as any user defined sections containing code or read-only data. The MPLAB XC8 compiler will produce a more accurate summary of Flash memory used than AVR-GCC when building

the same project. If you are comparing memory usage between the toolchains, ensure you check the map file to determine the exact size of the all sections used by your program.

5. Document Revision History

Revision A (November 2022)
Initial release of this document.

Microchip Information

The Microchip Website

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

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

Product Change Notification Service

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

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

Customer Support

Users of Microchip products can receive assistance through several channels:

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

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

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

Microchip Devices Code Protection Feature

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

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

Legal Notice

This publication and the information herein may be used only with Microchip products, including to design, test, and integrate Microchip products with your application. Use of this information in any other manner violates these terms. Information regarding device applications is provided only for your convenience and may be superseded

by updates. It is your responsibility to ensure that your application meets with your specifications. Contact your local Microchip sales office for additional support or, obtain additional support at www.microchip.com/en-us/support/design-help/client-support-services.

THIS INFORMATION IS PROVIDED BY MICROCHIP "AS IS". MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE, OR WARRANTIES RELATED TO ITS CONDITION, QUALITY, OR PERFORMANCE.

IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL, OR CONSEQUENTIAL LOSS, DAMAGE, COST, OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE INFORMATION OR ITS USE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO THE INFORMATION OR ITS USE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THE INFORMATION.

Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

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

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

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, Augmented Switching, BlueSky, BodyCom, Clockstudio, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, Espresso T1S, EtherGREEN, GridTime, IdealBridge, In-Circuit Serial Programming, ICSP, INICnet, Intelligent Paralleling, IntelliMOS, Inter-Chip Connectivity, JitterBlocker, Knob-on-Display, KoD, maxCrypto, maxView, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, RTAX, RTG4, SAM-ICE, Serial Quad I/O, simpleMAP, SimpliPHY, SmartBuffer, SmartHLS, SMART-I.S., storClad, SQI, SuperSwitcher, SuperSwitcher II, Switchtec, SynchroPHY, Total Endurance, Trusted Time, TSHARC, USBCheck, VariSense, VectorBlox, VeriPHY, ViewSpan, WiperLock, XpressConnect, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

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

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

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

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

ISBN: 978-1-6683-1447-0

Quality Management System

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

Worldwide Sales and Service

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