IAR AVR® 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

Notice to Development Tools Customers 1

1. Conventions Used in This Guide 5

2. Introduction 6

3. IAR Extended Keywords 7
   - 3.1. __eeprom Keyword 8
   - 3.2. __ext_io Keyword 9
   - 3.3. __far Keyword 9
   - 3.4. __farflash Keyword 10
   - 3.5. __farfunc Keyword 10
   - 3.6. __flash Keyword 11
   - 3.7. __generic Keyword 12
   - 3.8. __huge Keyword 12
   - 3.9. __hugeflash Keyword 13
   - 3.10. __interrupt Keyword 14
   - 3.11. __io Keyword 14
   - 3.12. __monitor Keyword 15
   - 3.13. __near Keyword 15
   - 3.14. __nearfunc Keyword 16
   - 3.15. __nested Keyword 16
   - 3.16. __no_alloc, __no_alloc16 Keywords 17
   - 3.17. __no_alloc_str, __no_alloc_str16 Keywords 17
   - 3.18. __no_init Keyword 17
   - 3.19. __no_runtime_init Keyword 18
   - 3.20. __noreturn Keyword 18
   - 3.21. __raw Keyword 18
   - 3.22. __regvar Keyword 19
   - 3.23. __ro_placement Keyword 20
   - 3.24. __root Keyword 20
   - 3.25. __task Keyword 21
   - 3.26. __tiny Keyword 21
   - 3.27. __tinyflash Keyword 21
   - 3.28. __version _x Keywords 22
   - 3.29. __x, __x_z, __z, __z_x Keywords 22

4. IAR Pragmas 24
   - 4.1. CX_LIMITED_RANGE Pragma 25
   - 4.2. FENV_ACCESS Pragma 25
   - 4.3. FP_CONTRACT Pragma 25
   - 4.4. basic_template_matching Pragma 25
   - 4.5. bitfields Pragma 25
   - 4.6. call_graph_root Pragma 25
   - 4.7. calls Pragma 26
   - 4.8. constseg Pragma 26
   - 4.9. data_alignment Pragma 26

4.10. dataseg Pragma....27
4.11. default_function_attributes Pragma....27
4.12. default_variable_attributes Pragma....28
4.13. diag_xxxx Pragmas....29
4.14. error Pragma....29
4.15. include_alias Pragma....29
4.16. inline Pragma....29
4.17. language Pragma....30
4.18. location Pragma....30
4.19. message Pragma....31
4.20. object_attribute Pragma....31
4.21. optimize Pragma....32
4.22. printf_args Pragma....32
4.23. public_equ Pragma....32
4.24. required Pragma....33
4.25. rtmodel Pragma....33
4.26. scanf_args Pragma....33
4.27. segment Pragma....33
4.28. type_attribute Pragma....33
4.29. vector Pragma....34
4.30. weak Pragma....34

5. IAR Intrinsic Functions....36
5.1. __delay_cycles Intrinsic Function....37
5.2. __DES_decryption Intrinsic Function....37
5.3. __DES_encryption Intrinsic Function....37
5.4. __disable_interrupt Intrinsic Function....38
5.5. __enable_interrupt Intrinsic Function....38
5.6. __extended_load_program_memory Intrinsic Function....39
5.7. __fractional_multiply_signed Intrinsic Function....39
5.8. __fractional_multiply_signed_with_unsigned Intrinsic Function....40
5.9. __fractional_multiply_unsigned Intrinsic Function....41
5.10. __get_interrupt_state Intrinsic Function....42
5.11. __indirect_jump_to Intrinsic Function....43
5.12. __insert_opcode Intrinsic Function....43
5.13. __lac Intrinsic Function....44
5.14. __las Intrinsic Function....45
5.15. __lat Intrinsic Function....46
5.16. __load_program_memory Intrinsic Function....46
5.17. __multiply_signed Intrinsic Function....47
5.18. __multiply_signed_with_unsigned Intrinsic Function....48
5.19. __multiply_unsigned Intrinsic Function....49
5.20. __no_operation Intrinsic Function....49
5.21. __require Intrinsic Function....50
5.22. __restore_interrupt Intrinsic Function....50
5.23. __reverse Intrinsic Function....51
5.24. __save_interrupt Intrinsic Function....51
5.25. __set_interrupt_state Intrinsic Function....51

5.26. __sleep Intrinsic Function....52
5.27. __swap_nibbles Intrinsic Function....53
5.28. __watchdog_reset Intrinsic Function....53
5.29. __xch Intrinsic Function....54

6. Document Revision History....55

Microchip Information....56
The Microchip Website....56
Product Change Notification Service....56
Customer Support....56
Microchip Devices Code Protection Feature....56
Legal Notice....56
Trademarks....57
Quality Management System....58
Worldwide Sales and Service....59

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	*IAR AVR® 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}`
Ellipses...	Replaces repeated text	*`var_name`* `[, `*`var_name`*`...]`

	Represents code supplied by user	`void main (void)`
		`{ ...`
		`}`

2. Introduction

This guide describes the source code changes that might be required should you decided to migrate a C-based project from the IAR C/C++ Compiler for AVR® (IAR) to the Microchip MPLAB® XC8 C Compiler (MPLAB XC8).

When migrating the project's source code, the majority of the required changes will be to non-standard keywords (referred to as extended keywords by IAR), pragmas, and built-in functions (referred to as intrinsic functions by IAR). Each of these IAR compiler features and the recommended migration steps are described in turn in the sections following. Differences in implementation-defined behavior between the two compilers are not considered here. The MPLAB XC8 compiler offers a different set of language extensions and features to those provided by the IAR compiler. These should be explored to take full advantage of the tool. Additionally, different command-line options are used to control how code is built.

A migrated project will produce a HEX file that differs to that built using the original project and the IAR 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 compiler and for more detailed information on the compiler's extensions and features.

3. IAR Extended Keywords

This section shows a summary of IAR extended keywords and the best MPLAB XC8 replacement where that is available. These keywords and their MPLAB XC8 equivalents are discussed in more detail in the sections that follow. It is recommended that you compare the relevant sections in the *MPLAB® XC8 C Compiler User's Guide for AVR MCUs* with those in your *IAR C/C++ Compiler User Guide* to ensure that migrated code will work as expected in all situations.

Table 3-1. MPLAB XC8 Equivalents to IAR Extended Keywords

IAR Keyword (links to explanatory section)	Suggested MPLAB XC8 Migration
`__eeprom`	Combined use of the `EEMEM` attribute and EEPROM-access functions
`__ext_io`	Use the `address` attribute
`__far`	No simple migration recommended
`__farflash`	Use either the `__memx` or `__flash`*n* qualifier
`__farfunc`	Remove the keyword
`__flash`	No migration necessary
`__generic`	Use the `__memx` qualifier
`__huge`	No simple migration recommended
`__hugeflash`	Either remove the keyword or use the `__memx` qualifier, based on the const-in-program-memory feature setting
`__interrupt`	Use the `__interrupt(`*number*`)` specifier
`__io`	Use one of the `io(`*n*`)`, `io_low(`*n*`)`, or `address(`*n*`)` attributes
`__monitor`	No simple migration recommended.
`__near`	Remove the keyword
`__nearfunc`	Remove the keyword
`__nested`	Use the `interrupt` attribute as well as the `__interrupt(`*n*`)` specifier
`__no_alloc, __no_alloc16`	No simple migration recommended
`__no_alloc_str, __no_alloc_str16`	No simple migration recommended
`__no_init`	Use the `__persistent` specifier or `persistent` attribute
`__no_runtime_init`	No simple migration recommended
`__noreturn`	No simple migration recommended
`__raw`	Consider the `naked` attribute
`__regvar`	Use the `asm("reg")` syntax and `register` keyword
`__ro_placement`	Remove the keyword when using the const-data-in-program-memory feature; otherwise, use the `__memx` qualifier
`__root`	No simple migration recommended
`__task`	No simple migration recommended
`__tiny`	Remove the keyword

.....continued	
IAR Keyword (links to explanatory section)	**Suggested MPLAB XC8 Migration**
`__tinyflash`	Use the `__flash` specifier
`__version1`, `__version2`, `__version4`	No simple migration recommended
`__x`, `__x_z`, `__z`, `__z_x`	No simple migration recommended

3.1 __eeprom Keyword

The IAR `__eeprom` keyword places objects declared with the attribute in EEPROM and additionally ensures that EEPROM-access routines are used to access such objects either directly or indirectly via any pointer qualified with this same keyword.

Suggested Replacement

There are two changes that must both be made to IAR code to have it behave in a similar way when using MPLAB XC8.

To place an object in EEPROM, include the `<avr/eeprom.h>` header and use the `EEMEM` attribute (or its expanded form, `__attribute__((section(".eeprom")))` with the object's definition. This attribute, however, does not affect how the object is accessed.

To read the object, use the `eeprom_read_byte()`, `eeprom_read_word()`, `eeprom_read_dword()`, or `eeprom_read_block()` functions as appropriate, and the corresponding `eeprom_write_xxxx()` functions to write to the object in EEPROM.

Caveats

None.

Examples

Consider migrating IAR code such as:

```
__eeprom int mode = 12;

volatile int x;
int main() {
  x = mode;
}
```

to MPLAB XC8 codes similar to:

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

int mode EEMEM = 12;

volatile int x;
int main() {
  x = eeprom_read_word(&mode);
}
```

Further Information

See the **Variables in EEPROM** section in the MPLAB XC8 C Compiler User's Guide for AVR MCUs for more information on these attributes.

3.2 __ext_io Keyword

The IAR `__ext_io` keyword specifies that the qualified object aliases memory-mapped SFRs, that is, the object is accessed from the AVR's IO registers. This is similar to the IAR `__io` attribute, except that it is meant for accessing SFRs above addresses 0x100.

Suggested Replacement
There is an MPLAB XC8 attributes that performs a similar tasks to this keyword, but there are some differences in its effect.

The MPLAB XC8 `address` attribute equates a symbol to an address, permitting the symbol to represent a register in the I/O space.

Caveats
The IAR `__ext_io` keyword ensures that memory is allocated to the object. That memory will be in the I/O space. When using the MPLAB XC8 `address` attribute, the compiler does not assign memory to the symbol; instead, it assumes that the symbol represents a peripheral register and merely equates the symbol to an address, which must be specified in the attribute.

Use of `__ext_io` implies the object is `volatile`. This is not the case when using the MPLAB XC8 `address` attribute.

Examples
Consider migrating IAR code such as:

```
__ext_io char TCB0_CTRLA;

int main(void) {
    volatile char x = TCB0_CTRLA;
}
```

to MPLAB XC8 code similar to:

```
volatile char TCB0_CTRLA __attribute__((address(0xB00)));

int main(void) {
    volatile char x = TCB0_CTRLA;
}
```

In this example, the compiler does not allocate memory at 0xB00; it merely assigns that address to the `TCB0_CTRLA` identifier.

Further Information
See the **Attributes** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on this attribute.

3.3 __far Keyword

The IAR `__far` keyword places and accesses objects in far data memory, which has an addressable range 0x0-0xFFFFFF. Pointers qualified with this keyword can point to any address in this range.

Suggested Replacement
None. MPLAB XC8 only supports 16-bit data memory pointers.

3.4 __farflash Keyword

The IAR `__farflash` keyword places objects in program memory, and uses 24-bit addresses to read such objects either directly or indirectly via any pointer qualified with this same keyword. Arithmetic on 24-bit wide pointers is only performed on the lower 16 bits, except for comparisons, which are always performed on the entire address.

Suggested Replacement
This keyword can be removed when using a memory-placement feature of MPLAB XC8, or alternatively, there are several MPLAB XC8 qualifiers that performs a similar tasks to this keyword, but there are some differences in their effect.

When enabled (the default state), the MPLAB XC8's const-data-in-program-memory feature places `const` objects into program memory. In this case, the IAR keyword is redundant and should be removed.

When this feature is disabled, using the `-mno-const-data-in-progmem` option, use the `const` qualifier provided by MPLAB XC8 and either the `__memx` qualifier or one of the `__flash`*n* qualifiers, where *n* is the 64 KB flash segment in which to place and access the object. When using the `__memx` qualifier, both access and address arithmetic is on a full 24 bit address. With any of the flash qualifiers, pointers and address arithmetic is 16-bit, and the compiler sets RAMPZ to the appropriate 64 KB segment number on each access.

Caveats
The `const` qualifier must be used with either `__memx` or any of the flash qualifiers when using MPLAB XC8, whereas this is only preferable when using `__farflash` with IAR.

If the const-in-program-memory feature is disabled, alternate versions of the string functions normally provided by `<string.h>` must be used to access string objects located in program memory.

Examples
Consider migrating IAR code such as:

```
__farflash int x = 200; /* Place and access x somewhere in flash */
volatile int y;
int main() {
    y = x;
}
```

to MPLAB XC8 code similar to:

```
const volatile __flash1 int x = 200; /* Place and access x in the 64K-128K segment
of flash */
volatile int y;
int main(void) {
    y = x;
}
```

Further Information
See the **Special Type Qualifiers** and **Options Specific to AVR Devices** sections in the MPLAB XC8 C Compiler User's Guide for AVR MCUs for more information on these qualifiers and option.

3.5 __farfunc Keyword

The IAR `__farfunc` keyword places the qualified function within program memory, and when used with the definition of a function pointer, the pointer is made 24 bits wide so that it can access functions anywhere in the range 0x0-FFFFFF.

Suggested Replacement
The keyword can be removed.

When using MPLAB XC8, the `-mrelax` option automatically handles placement and indirect calls for functions located anywhere in program memory. Function pointers are always 16-bit wide, but on devices with more than 128 KB of program memory, they point into a trampoline table placed in near flash. The trampoline table entries then jump to the actual function using a `jmp` instruction.

Caveats
When function addresses are obtained without a function symbol, for example, when casting an integer constant to a function pointer, this method might not have worked as expected. For example:

```
int main(void)
{
    ((int(*)(void)) 0x20000)();    /* will not work */
}
```

might fail. Instead, do the following:

```
int main(void)
{
    extern int foo(void); /* Force creation of a symbol, with address set at link
time */
    foo();
}
```

and associate an address with the symbol using the `xc8-cc` driver option `-Wl,-defsym,foo=0x40000`, for example.

Examples
Consider migrating IAR code such as:

```
__farfunc void incMode(void);
```

to MPLAB XC8 code similar to:

```
void incMode(void);
```

Further Information
See the **Function Pointers** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on how functions are handled.

3.6 __flash Keyword

The IAR `__flash` keyword places objects in the first 64 KB segment of program memory and when used with the definition of a pointer, that pointer can access objects in this same segment.

Suggested Replacement
There is an identical MPLAB XC8 qualifier; alternatively, the qualifier can be removed when an MPLAB XC8 memory-placement feature is in effect.

The MPLAB XC8 compiler implements a `__flash` keyword with the same behavior, so this IAR keyword does not need to be removed or replaced.

Alternatively, when enabled (the default state), the MPLAB XC8's const-data-in-program-memory feature places `const` objects into program memory. In this case, the IAR keyword is redundant and can be removed.

Caveats
The `const` qualifier must be used with the `__flash` qualifier when using MPLAB XC8, whereas this is only preferable when using `__flash` with IAR.

Further Information
See the **Special Type Qualifiers** and **Options Specific to AVR Devices** sections in the MPLAB XC8 C Compiler User's Guide for AVR MCUs for more information on this qualifier and option.

3.7 __generic Keyword

The IAR `__generic` keyword is used with pointer definitions to allow those pointers to accesses objects from both data and program spaces, based on the MSb of the address. The pointer size varies with the selected device.

Suggested Replacement
There are MPLAB XC8 qualifiers that performs similar tasks to this keyword, but there are some differences in their effect.

If the const-in-program-memory feature is enabled (the default state), use the standard `const` qualifier provided by MPLAB XC8 with a pointer definition on devices which do not map their program memory in the data space to make that pointer 24-bits wide.

If this feature is disabled, using the `-mno-const-data-in-progmem` option, use both the `const` and `__memx` qualifiers with a pointer definition. In either case, the MSb of the pointer's address is used to determine whether to read from the data or program address spaces. Such pointers can access the lower 64 KB of the data space.

Caveats
If the const-in-program-memory feature is disabled, alternate versions of the string functions normally provided by `<string.h>` must be used to access string objects located in program memory.

Examples
Consider migrating IAR code such as:

```
const __generic char * p;
volatile char x;
int main(void) {
    x = *p;
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

const __memx char * p;
volatile char x;
int main(void) {
    x = *p;
}
```

Further Information
See the **Special Type Qualifiers** and **Options Specific to AVR Devices** sections in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on this qualifier and option.

3.8 __huge Keyword

The IAR `__huge` keyword places and accesses objects in huge data memory (addressable range 0x0-0xFFFFFF). Pointer arithmetic and access is all 24-bit. Pointers qualified with this keyword can point to any address in this range.

Suggested Replacement
None. MPLAB XC8 only supports 16-bit data memory pointers.

3.9 __hugeflash Keyword

The IAR `__hugeflash` keyword places and accesses objects in huge program memory (addressable range 0x0-0x7FFFFF). Pointer arithmetic and access is all 24-bit. Pointers qualified with this keyword can point to any address in this range.

Suggested Replacement

There are two ways of having MPLAB XC8 code perform a similar task to this keyword.

When the `-mconst-data-in-progmem` const-in-program-memory feature is in effect (the default operation), this IAR keyword can simply be removed. In this case, all `const`-qualified objects are automatically placed in and read from program memory, and 24-bit wide pointers are used.

When the `-mno-const-data-in-progmem` option has been used to disabled the const-in-program-memory feature, use the MPLAB XC8 `__memx` qualifier, which places `const` objects in flash and accesses these using 24-bit addresses.

Caveats

The `const` qualifier must be used with the `__memx` qualifier when using MPLAB XC8, whereas this is only preferable when using `__hugeflash` with IAR.

Access of objects using `__memx` will be slightly slower than those using a 24-bit flash-only pointer, as `__memx` addresses represent a combined flash + data memory address space.

Examples

Consider migrating IAR code such as:

```
__hugeflash char arr[] = "abc";

volatile int x;
int main(void) {
    x = arr[x];
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

const char arr[] = "abc";

volatile int x;
int main(void) {
    x = arr[x];
}
```

when the `-mno-const-data-in-progmem` const-in-program-memory feature is enabled, or to:

```
#include <xc.h>

const __memx char arr[] = "abc";

volatile int x;
int main(void) {
    x = arr[x];
}
```

when the const-in-program-memory feature has been disabled.

Further Information

See the **Special Type Qualifiers** and **Options Specific to AVR Devices** sections in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on this qualifier and option.

3.10 __interrupt Keyword

The IAR `__interrupt` keyword specifies the qualified function is an interrupt handler. The interrupt vector is specified separately with a `#pragma vector = number` pragma that must precede the function.

Suggested Replacement
There is an MPLAB XC8 qualifier that performs a similar task to this keyword.

Use the MPLAB XC8 `__interrupt` qualifier with the required vector number as argument. The compiler provides predefined macros that represent the vector numbers, for example `SPI_STC_vect_num`.

Caveats
The Common C Interface must be enabled using `-mext=cci` to use the `__interrupt` qualifier.

Examples
Consider migrating IAR code such as:

```
volatile int x;
#pragma vector=2
void __interrupt incIsr(void) {
    x++;
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

volatile int x;
void __interrupt(2) incIsr(void) {
    x++;
}
```

Further Information
See the **Interrupts** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on writing interrupt functions.

3.11 __io Keyword

The IAR `__io` keyword specifies that the object aliases memory mapped SFRs and that access will be performed on the AVR's IO registers. This keyword is used for accessing SFRs at addresses below 0x100.

Suggested Replacement
There are several MPLAB XC8 attributes that perform similar tasks to this keyword, but there are some differences in their usage and effect.

Use one of `__attribute__((io_low(n)))`, `__attribute__((io(n)))` or `__attribute__((address((n))))`, depending on the desired address of the SFR, where `n` is the mapped address in data memory. If the SFR is in the bit-addressable IO range (IO memory address 0x0-0x1F), then use the `io_low` attribute. If the SFR is in the IN/OUT range (IO memory address 0x20-0x3F), then use the `io` attribute; otherwise, use the `address` attribute.
The compiler will use the `in` and/or `out` instructions whenever possible to access the symbol.

Caveats
An object specified with this IAR keyword is assigned memory, like with any ordinary definition; however, this keyword ensures that this memory is in the I/O memory space. By comparison, MPLAB XC8 does not assign memory to the symbol that uses these attributes, assuming that the symbol represents a peripheral register. The symbol defined with this attribute is merely equated with an address.

A warning will be issued if the declaration for a symbol using either the `io` or `io_low` attributes is not `volatile`.

Examples
Consider migrating IAR code such as:

```
__io int userMode;
```

to MPLAB XC8 codes similar to:

```
// declare userMode as being at the specified address in upper I/O memory
volatile int userMode __attribute__((io(0x42)));

// or declare userMode as being in the lower area in I/O memory
volatile int userMode __attribute__((io_low(0x28)));

// or declare userMode as being at the specified address in I/O memory
volatile int userMode __attribute__((address(0x50)));
```

Further Information
See the **Attributes** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on these attributes.

3.12 __monitor Keyword

The IAR `__monitor` keyword causes interrupts to be disabled during execution of the specified function, providing protection for operations that must be performed atomically.

Suggested Replacement
None.

3.13 __near Keyword

The IAR `__near` keyword places qualified objects in near memory (0-64 KB). Pointers qualified with this keyword are 16-bits wide and can point to any address in this range.

Suggested Replacement
The keyword can be removed.

When using MPLAB XC8, data memory pointers are always 16-bits wide, and only 0-64 KB of data memory can be addressed.

Examples
Consider migrating IAR code such as:

```
__near int mode;

volatile int x;
int main(void) {
  x = mode;
}
```

to MPLAB XC8 code similar to:

```
int mode;

volatile int x;
int main(void) {
  x = mode;
}
```

3.14 __nearfunc Keyword

The IAR `__nearfunc` keyword places specified function in first 128 KB of program memory. When used with the definition of a function pointer, the pointer is made 16 bits wide so that it can access functions any function in this range.

Suggested Replacement
The keyword can be removed.

Function pointers are always 16 bits wide, but there is no guarantee that the function will be placed in the first 128 KB of program memory. However, when using MPLAB XC8, the `-mrelax` option automatically handles placement and indirect calls for functions, regardless of where they are located in program memory. On devices with more than 128 KB of program memory, function addresses point into a trampoline table placed in near flash. The trampoline table entries then jump to the actual function using a `jmp` instruction.

Caveats
When function addresses are obtained without a function symbol, for example when casting an integer constant to a function pointer, this method might not worked as expected. For example:

```
int main(void)
{
    ((int(*)(void)) 0x20000)();    /* will not work */
}
```

might fail. Instead, do the following:

```
int main(void)
{
    extern int foo(void); /* Force creation of a symbol, with address set at link
time */
    foo();
}
```

and associate an address with the symbol using the `xc8-cc` driver option `-Wl,-defsym,foo=0x40000`, for example.

Examples
Consider migrating IAR code such as:

```
__nearfunc void incMode(void);
```

to MPLAB XC8 code similar to:

```
void incMode(void);
```

Further Information
See the **Function Pointers** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on how functions are handled.

3.15 __nested Keyword

The IAR `__nested` keyword enables interrupts in the prologue of an interrupt handler, that is it allows nested interrupts to occur.

Suggested Replacement
There is an MPLAB XC8 attribute and specifier that together perform a similar tasks to this keyword.

Use the MPLAB XC8 `interrupt` attribute in addition to the `__interrupt(`*`number`*`)` specifier. The `ISR_NOBLOCK` macro can be used as an alias for `__attribute__((interrupt))` if preferred.

Caveats
The Common C Interface must be enabled using `-mext=cci` to use the `__interrupt` qualifier.

Examples
Consider migrating IAR code such as:

```
volatile int x;
#pragma vector=2
void __nested __interrupt incIsr(void) {
    x++;
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

volatile int x;
void __attribute__((interrupt)) __interrupt(2) incIsr(void) {
    x++;
}
```

Further Information
See the **Interrupts** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on writing interrupt functions.

3.16 __no_alloc, __no_alloc16 Keywords

The IAR `__no_alloc` and `__no_alloc16` keywords specify that the qualified constant should be placed in the ELF file without being part of the linked application.

Suggested Replacement
None. This feature is not supported.

3.17 __no_alloc_str, __no_alloc_str16 Keywords

The IAR `__no_alloc_str` and `__no_alloc_str16` keywords specify that the qualified string literal should be placed in the ELF file without being part of the linked application.

Suggested Replacement
None. This feature is not supported.

3.18 __no_init Keyword

The IAR `__no_init` keyword specifies that the qualified object is not initialized by the runtime startup code.

Suggested Replacement
There is a MPLAB XC8 specifier that performs a similar task.

Use the `__persistent` specifier or (or its expanded form, `__attribute__((persistent))` with the object's definition.

Caveats
The Common C Interface must be enabled using `-mext=cci` to use the `__persistent` specifier form.

Examples
Consider migrating IAR code such as:

```
volatile __no_init int x;
int main(void) {
    x = 2;
    return 0;
}
```

to MPLAB XC8 codes similar to:

```
#include <xc.h>

volatile __persistent int x;
int main(void) {
    x = 2;
    return 0;
}
```

Further Information
See the **Attributes** and **Options for Controlling the C Dialect** sections in the MPLAB XC8 C Compiler User's Guide for AVR MCUs for more information on this option and attribute.

3.19 __no_runtime_init Keyword

The IAR `__no_runtime_init` keyword specifies that the qualified object should be initialized by programming of the device, not by the runtime startup code.

Suggested Replacement
None. This feature is not supported.

3.20 __noreturn Keyword

The IAR `__noreturn` keyword specifies that the qualified function does not return.

Suggested Replacement
None.

3.21 __raw Keyword

The IAR `__raw` keyword prevents saving call-used registers in interrupt functions.

Suggested Replacement
There is no MPLAB XC8 feature that performs the same function as this keyword; however, there is an attribute which performs a similar task.

The `naked` attribute can be used to prevent generation of the entire prologue and epilogue context switching associated with an interrupt function.

Caveats
The `naked` attribute prevents the call-used registers from being saved, but it also omits code that performs other operations typically required by interrupt functions, such as generation of the `reti` instruction, and saving and clearing r1 (`__zero_reg__`) on entry, and the reverse process on exit.

Examples
Consider migrating IAR code such as:

```
volatile int x;

#pragma vector=2
__raw __interrupt void incIsr(void) {
    x++;
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

volatile int x;
void __attribute__((naked)) __interrupt(2) incIsr(void) {
    // insert hand-written context switch code here
    x++;
    // insert hand-written context switch and return-from-interrupt code here
}
```

Further Information
See the **Interrupts** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on writing interrupt functions.

3.22 __regvar Keyword

The IAR `__regvar` keyword places static storage duration variables in registers or register sets.

Suggested Replacement
There is an MPLAB XC8 feature that performs a similar tasks to this keyword, but there are some differences in its effect.

Use the `asm("reg")` syntax along with the standard `register` keyword when defining an object to place that object in a register or in consecutive registers. Note that the compiler will issue an error if an attempt is made to reserve critical registers (like argument registers or the frame pointer register) using this option.

Caveats
The compiler might use the registers assign in different translation units.

Examples
Consider migrating IAR code such as:

```
__regvar __no_init int myVar @ 14;

volatile int x;
int main() {
    x = myVar;
}
```

and the `--lock_regs 2`IAR compiler option to MPLAB XC8 code similar to:

```
register int myVar asm("r14");

volatile int x;
int main() {
    x = myVar;
}
```

Further Information
See the **Register Usage** and **Variables in Registers** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on writing interrupt functions.

3.23 __ro_placement Keyword

The IAR `__ro_placement` keyword specifies that the qualified `const volatile` objects should be placed in read-only memory.

Suggested Replacement
This keyword can be removed when using a memory-placement feature of MPLAB XC8, or alternatively, there are MPLAB XC8 qualifiers that perform a similar task to this keyword.

When enabled (the default state), the MPLAB XC8's const-data-in-program-memory feature places `const volatile` objects into program memory. In this case, the IAR keyword is redundant and should be removed.

When this feature is disabled, using the `-mno-const-data-in-progmem` option, use both the `const` and `__memx` qualifiers to have these objects placed in program memory.

Caveats
If the const-in-program-memory feature is disabled, alternate versions of the string functions normally provided by `<string.h>` must be used to access string objects located in program memory.

Examples
Consider migrating IAR code such as:

```
volatile const int __ro_placement myVar = 2;

volatile int x;
int main(void) {
    x = myVar;
}
```

to MPLAB XC8 code similar to:

```
volatile const int myVar = 2;

volatile int x;
int main(void) {
    x = myVar;
}
```

when building with the const-data-in-program-memory feature enabled.

Further Information
See the **Options Specific to AVR Devices** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on the const-data-in-program-memory feature.

3.24 __root Keyword

The IAR `__root` keyword ensures that qualified objects or functions are not optimized away, even if they have not been used.

Suggested Replacement
None.

3.25 __task Keyword

The IAR `__task` keyword omits the saving and restoration of callee-saved registers in the specified function. It is intended to be used for start function of an RTOS task.

Suggested Replacement
None.

3.26 __tiny Keyword

The IAR `__tiny` keyword places and accesses objects in tiny data memory, which has an addressable range 0x0-FF. Pointers qualified with this keyword are 8 bits wide and can point to any address in this range.

Suggested Replacement
Remove the keyword, but note the different operation of such code.

MPLAB XC8 does not provide support for 8-bit data pointers. Without the `__tiny` keyword, MPLAB XC8 will use 16-bit pointers and addresses to access objects.

Caveats
Programs will execute correctly with this keyword removed, unless the program makes explicit assumptions that addresses are 8-bits wide.

Examples
Consider migrating IAR code such as:

```
__tiny int mode;

volatile int x;
int main(void) {
  x = mode;
}
```

to MPLAB XC8 code similar to:

```
int mode;

volatile int x;
int main(void) {
  x = mode;
}
```

3.27 __tinyflash Keyword

The IAR `__tinyflash` keyword places and accesses objects from program memory in the address range of 0-0xFF, using 8-bit pointers., which has an addressable range 0x0-FF. Pointers qualified with this keyword are 8 bits wide and can point to any address in this range.

Suggested Replacement
There is an MPLAB XC8 specifier that performs a similar tasks to this keyword, but there are some differences in its effect.

Use the `__flash` specifier to have objects placed in program memory.

Caveats
This MPLAB XC8 specifier will place objects in the first 64 KB of program memory and pointers and addresses to this space will be 16 bits wide. Programs using this specifier will execute correctly, unless the program makes explicit assumptions that addresses are 8-bits wide.

The `const` qualifier must be used with the `__flash` qualifier when using MPLAB XC8, whereas this is not necessary when using `__flash` with IAR.

Examples
Consider migrating IAR code such as:

```
__tinyflash int mode;

volatile int x;
int main(void) {
  x = mode;
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

const __flash int mode;

volatile int x;
int main(void) {
  x = mode;
}
```

Further Information
See the **Special Type Qualifiers** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on this qualifier.

3.28 __version _x Keywords

The IAR `__version_1` and `__version_2` keywords specify backward-compatible IAR calling conventions when calling assembler functions from C; `__version_4` is the default calling convention.

Suggested Replacement
None. These calling conventions are not available.

Further Information
See the **Functions** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on how function calls are encoded.

3.29 __x, __x_z, __z, __z_x Keywords

The IAR `__x` keyword places the first pointer of the parameter list in the X register, and with the `__x_z` keyword, places the second pointer in the register Z.

The IAR `__z` keyword places the first pointer of the parameter list in the Z register, and with the `__z_x` keyword, places the second pointer in the register X.

Suggested Replacement
None. These calling conventions are not available.

Further Information
See the **Functions** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on how function calls are encoded.

4. IAR Pragmas

This section shows a summary of IAR pragmas and the best MPLAB XC8 replacement where that is available. These pragmas and their MPLAB XC8 equivalents are discussed in more detail in the sections that follow. It is recommended that you compare the relevant sections in the *MPLAB® XC8 C Compiler User's Guide for AVR MCUs* with those in your *IAR C/C++ Compiler User Guide* to ensure that migrated code will work as expected in all situations.

Table 4-1. MPLAB XC8 Equivalents to IAR Pragmas

IAR Pragma (links to explanatory section)	Suggested MPLAB XC8 Migration
`CX_RANGE_LIMITED`	The pragma can be removed
`FENV_ACCESS`	The pragma can be removed
`FP_CONTRACT`	No simple migration recommended
`basic_template_matching`	No simple migration recommended
`bitfields`	No simple migration recommended
`call_graph_root`	No simple migration recommended
`calls`	No simple migration recommended
`constseg`	Use the `section` attribute
`data_alignment`	Use the `aligned` attribute
`dataseg`	Use the `section` attribute
`default_function_attributes`	Consider the `section` attribute
`default_variable_attributes`	Consider the `typedef` specifier and `section` attribute
`diag_xxxx`	No simple migration recommended
`error`	No simple migration recommended
`include_alias`	No simple migration recommended
`inline`	Use the `always_inline` or `noinline` attribute
`language`	No simple migration recommended
`location`	Use the `__at` specifier and `section` attribute
`message`	No simple migration recommended
`object_attribute`	Migrate individual attributes specified by this pragma
`optimize`	No simple migration recommended
`printf_args`	No simple migration recommended
`public_equ`	Use the `asm()` statement
`required`	No simple migration recommended
`rtmodel`	No simple migration recommended
`scanf_args`	No simple migration recommended
`segment`	The pragma can be removed
`type_attribute`	Migrate individual attributes specified by this pragma

.....continued	
IAR Pragma (links to explanatory section)	**Suggested MPLAB XC8 Migration**
vector	Use the `__interrupt(number)` specifier
weak	Use the `weak` function attribute

4.1 CX_LIMITED_RANGE Pragma

The IAR `CX_LIMITED_RANGE` standard C pragma specifies that the compiler can use the normal complex mathematic formulas for multiplication, division, and `abs()`. It is ignored by the IAR compiler.

Suggested Replacement
The pragma can be removed. MPLAB XC8 for AVR does not support `<complex.h>`.

4.2 FENV_ACCESS Pragma

The IAR `FENC_ACCESS` standard C pragma specifies whether source code accesses the floating-point environment.

Suggested Replacement
The pragma can be removed. MPLAB XC8 for AVR ignores this pragma.

4.3 FP_CONTRACT Pragma

The IAR `FP_CONTRACT` standard C pragma specifies whether the compiler can contract floating-point expressions. Only the `ON` argument value is supported by the IAR compiler.

Suggested Replacement
None.

4.4 basic_template_matching Pragma

The IAR `basic_template_matching` pragma controls C++ template matching for memory attributes like `__near`.

Suggested Replacement
None.

4.5 bitfields Pragma

The IAR `bitfields` pragma controls the memory ordering of bit-fields within a structure.

Suggested Replacement
None.

4.6 call_graph_root Pragma

The IAR `call_graph_root` pragma specifies that the following function is a call graph root of an arbitrary category for the purposes of stack usage analysis.

Suggested Replacement
None. The MPLAB XC8's stack usage analysis will deduce call graph roots, but there is no option to arbitrarily change or set categories of call graph roots.

4.7 calls Pragma

The IAR `calls` pragma specifies the possible target functions that could be invoked in the indirect call in the following statement.

Suggested Replacement
None. The MPLAB XC8 stack guidance feature reports indirect calls as a caution.

4.8 constseg Pragma

The IAR `constseg` pragma changes the segment in which `const` data is placed. Any arbitrary named segment can be provided, and default restores placement to the default segment.

Suggested Replacement
There is no equivalent MPLAB XC8 pragma, but there is an attribute that performs a similar task.

The MPLAB XC8 `section` attribute can be used with a `const` object to specify the section in which it is placed. If the Common C Interface is enabled, you can instead use the `__section` specifier.

Caveats
The sections generated by this attribute are not concatenated across translation units by the linker, and the `-Wl,--section-start,`*`section_name=address`* option, which can place a section in memory, only works with custom (non-standard) sections.

Examples
Consider migrating IAR code such as:

```
#pragma constseg=HI_MARK
const int factorySettings[] = {40, 51, 127, 0};
#pragma constseg=default
```

to MPLAB XC8 code similar to:

```
#include <xc.h>
const int __section("HI_MARK") factorySettings[] = {42, 15, -128, 0};
```

and build with the `-mext=cci` option.

Further Information
See the **Attributes** and **Ext Option** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on this attribute.

4.9 data_alignment Pragma

The IAR `data_alignment` pragma controls the memory alignment of the variable immediately following.

Suggested Replacement
There is no equivalent MPLAB XC8 pragma, but there is an attribute that performs a similar task.

The MPLAB XC8 `aligned(`*`n`*`)` attribute aligns the qualified object's address with the next address that is a whole multiple of the numerical value *`n`*. If the CCI is enabled, a more portable macro, `__align(`*`n`*`)` (note the different spelling), is available. This attribute works with automatic as well as static storage duration objects.

Caveats
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.

Examples
Consider migrating IAR code such as:

```
#pragma data_alignment=4
char marker = 3;
```

to MPLAB XC8 code similar to:

```
char marker __attribute__((aligned(4))) = 3;
```

Further Information
See the **Attributes** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on this attribute.

4.10 dataseg Pragma

The IAR `dataseg` pragma changes the segment in which objects are placed. Any arbitrary named segment can be provided, and default restores placement to the default segment.

Suggested Replacement
There is no equivalent MPLAB XC8 pragma, but there is an attribute that performs a similar task.

The MPLAB XC8 `section` attribute can be used with objects to specify the section in which it is placed. If the Common C Interface is enabled, you can instead use the `__section` specifier.

Caveats
The sections generated by this attribute are not concatenated across translation units by the linker, and the `-Wl,--section-start,`*`section_name=address`* option, which can place a section in memory, only works with custom (non-standard) sections.

Examples
Consider migrating IAR code such as:

```
#pragma dataseg=US_SEG
int swOffset;
#pragma dataseg=default
```

to MPLAB XC8 code similar to:

```
#include <xc.h>
int __section("US_SEG") swOffset;
```

and build with the `-mext=cci` option.

Further Information
See the **Attributes** and **Ext Option** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on this attribute.

4.11 default_function_attributes Pragma

The IAR `default_function_attributes` pragma sets default segment placement, type attributes, and object attributes for function declarations and definitions that do not otherwise specify type or object attributes or location.

Suggested Replacement
There is no equivalent MPLAB XC8 pragma. Some of this pragma's functionality can be achieved with an attribute.

The MPLAB XC8 `section` attribute can be used with functions to specify the section in which it is placed. If the Common C Interface is enabled, you can instead use the `__section` specifier.

Caveats
The sections generated by this attribute are not concatenated across translation units by the linker, and the `-Wl,--section-start,`*`section_name=address`* option, which can place a section in memory, only works with custom (non-standard) sections.

Examples
Consider migrating IAR code such as:

```
#pragma default_function_attributes = @ "SP_SEG"
int inc(int x)
{
    return x + 1;
}
#pragma default_function_attributes =
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

int __section("SP_SEG") inc(int x)
{
    return x + 1;
}
```

and build with the `-mext=cci` option.

Further Information
See the **Attributes** and **Ext Option** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on this attribute.

4.12 default_variable_attributes Pragma

The IAR `default_variable_attributes` pragma sets default segment placement, type attributes, and object attributes for static storage duration object declarations and definitions that do not otherwise specify type or object attributes or location.

Suggested Replacement
There is no equivalent MPLAB XC8 pragma. Some of this pragma's functionality can be achieved with a typedef or attribute.

The `typedef` storage-class specifier can be used to associate qualifiers, like `__memx`, with a type that can be used with the objects.

The MPLAB XC8 `section` attribute can additionally be used with objects to specify the section in which it is placed. If the Common C Interface is enabled, you can instead use the `__section` specifier.

Caveats
The sections generated by this attribute are not concatenated across translation units by the linker, and the `-Wl,--section-start,`*`section_name=address`* option, which can place a section in memory, only works with custom (non-standard) sections.

Examples
Consider migrating IAR code such as:

```
#pragma default_function_attributes = @ "MYSEG" __farflash
const int startP  = 20;
#pragma default_function_attributes =
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

typedef const volatile __memx int cm_t;
cm_t startP __section("MYSEG") = 20;
```

and build with the `-mext=cci` option.

Further Information
See the **Attributes** and **Ext Option** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on this attribute.

4.13 diag_xxxx Pragmas

The IAR `diag_default`, `diag_error`, `diag_remark`, `diag_suppress`, and `diag_warning` pragmas changes the severity level for the specified diagnostic messages.

Suggested Replacement
None.

4.14 error Pragma

The IAR `error` pragma triggers a compile-time error when parsed. It can trigger an error when a preprocessor macro is used by using the `_Pragma` operator.

Suggested Replacement
None.

4.15 include_alias Pragma

The IAR `include_alias` pragma triggers a compile-time error when parsed. It can trigger an error when a preprocessor macro is used by using the `_Pragma` operator.

Suggested Replacement
None.

4.16 inline Pragma

The IAR `inline` pragma ensures that the following function is either in-lined or not in-lined, based on the pragma parameter.

Suggested Replacement
There is no equivalent MPLAB XC8 pragma, but there are attributes that perform similar tasks.

Use the `always_inline` attribute to ensure a function is in-lined; use the `noinline` attribute to prevent in-lining from taking place.

Caveats
When indirectly calling a function using the `always_inline` attribute, the compiler might or might not in-line it depending on the current optimization level, and a failure to in-line such a call might or might not be reported.

Examples
Consider migrating IAR code such as:

```
volatile int x;
#pragma inline=forced
void foo(void) {
    x++;
}


#pragma inline=never
void __attribute__((noinline)) bar(void) {
    x--;
}

int main() {
    foo();
    bar();
    return 0;
}
```

to MPLAB XC8 code similar to:

```
volatile int x;
void __attribute__((always_inline)) foo(void) {
    x++;
}

void __attribute__((noinline)) bar(void) {
    x--;
}

int main() {
    foo();
    bar();
    return 0;
}
```

Further Information
See the **Function Specifiers** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on these specifiers.

4.17 language Pragma

The IAR `language` pragma controls IAR language extensions.

Suggested Replacement
None. Remove the pragma and ensure that any IAR language extension has been migrated to the MPLAB XC8 equivalent, where possible.

4.18 location Pragma

The IAR `location` pragma places the following static storage duration object at the specified address or in the named section, based on the pragma argument.

Suggested Replacement
There is no equivalent MPLAB XC8 pragma, but there are two specifiers that perform similar tasks.

Use the `__at(address)` specifier to have an object placed at a specific address.

The MPLAB XC8 `section` attribute can be used with objects to specify the section in which it is placed. If the Common C Interface is enabled, you can instead use the `__section` specifier.

Caveats
The Common C Interface must be enabled using `-mext=cci` and `<xc.h>` must be included into your code to use the `__at()` specifier.

The sections generated by the `section` attribute are not concatenated across translation units by the linker, and the `-Wl,--section-start,section_name=address` option, only works with custom (non-standard) sections.

Examples
Consider migrating IAR code such as:

```
#pragma location=0x200
__no_init volatile int myVar;   /* myVar is located at address 0x200 */

#pragma segment="FLASH"
#pragma location="FLASH"
__no_init char arr2[];           /* arr2 is located in segment FLASH */

int main(void) {
    myVar = arr[myVar] + arr2[myVar];
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

const char arr[] __at(0x500) = "abc";  /* Place at address 0x500 in program memory
*/
volatile int myVar __at(0x800200) = 3; /* Place at address 0x200 in data memory */

const char arr2[] __section("FLASH") = "def"; /* Placed in named section mysec */
int main(void) {
    myVar = arr[myVar] + arr2[myVar];
}
```

and build with the `-mext=cci` option.

Further Information
See the **Absolute Variables**, **Attributes** and **Ext Option** sections in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on these specifiers, attribute, and option.

4.19 message Pragma

The IAR `message` pragma prints the specifier string argument as a message at compile time.

Suggested Replacement
None.

4.20 object_attribute Pragma

The IAR `object_attribute` pragma sets object attributes to the following function or object.

Suggested Replacement
Remove the pragma and ensure that any IAR object attributes specified with this pragma have been migrated to the MPLAB XC8 equivalent and attached to the definition of the object or function, where possible.

Caveats
None

Examples
Consider migrating IAR code such as:

```
#pragma object_attribute=__no_init
char maker = 3;
```

to MPLAB XC8 code similar to:

```
__persistent char maker = 3;
```

4.21 optimize Pragma

The IAR `optimize` pragma decreases the optimization level, or turns off specific optimizations for the function that follows.

Suggested Replacement
None.

4.22 printf_args Pragma

The IAR `printf_args` pragma verifies the arguments of any calls to the following printf-style function against the specifiers in the format string.

Suggested Replacement
None.

4.23 public_equ Pragma

The IAR `public_equ` pragma defines a public assembler symbol with the specified value.

Suggested Replacement
There is no equivalent MPLAB XC8 pragma, but a similar task can be performed by adding in-line assembly.

Use the MPLAB XC8 `asm` statement add the label and equate it to a value.

Caveats
None

Examples
Consider migrating IAR code such as:

```
#pragma public_equ="mySymbol",0x1000
```

to MPLAB XC8 code similar to:

```
asm(".global sym\r\nmySymbol = 0x1000");
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on adding in-line assembly.

4.24 required Pragma

The IAR `required` pragma ensures a symbol is output, even if it appears to be unreferenced.

Suggested Replacement
None.

4.25 rtmodel Pragma

The IAR `rtmodel` pragma adds a runtime model key/value to a module, which can be used by the linker to check consistency between modules.

Suggested Replacement
None.

4.26 scanf_args Pragma

The IAR `scanf_args` pragma verifies the arguments of any calls to the following scanf-style function against the specifiers in the format string.

Suggested Replacement
None.

4.27 segment Pragma

The IAR `segment` pragma defines a new named segment with specified memory attributes and alignment.

Suggested Replacement
The pragma can be removed.

Objects and functions can be placed in user-defined sections by using the `section` attribute (for example, see the suggested replacement at 4.8. constseg Pragma), but sections do not need to be defined separately.

4.28 type_attribute Pragma

The IAR `type_attribute` pragma sets type attributes to the following function or object.

Suggested Replacement
Remove the pragma and ensure that any IAR type attributes specified with this pragma have been migrated to the MPLAB XC8 equivalent and attached to the definition of the object or function, where possible.

Caveats
None

Examples
Consider migrating IAR code such as:

```
#pragma object_attribute=__tinyflash
const char maker = 3;
```

to MPLAB XC8 code similar to:

```
__flash const char maker = 3;
```

4.29 vector Pragma

The IAR `vector` pragma sets the interrupt number associated with the following interrupt handler function.

Suggested Replacement
Remove the pragma and ensure that the vector number has been specified as the argument to the `__interrupt` qualifier used with the interrupt function.

Caveats
None

Examples
Consider migrating IAR code such as:

```
volatile int x;
#pragma vector=2
void __interrupt incIsr(void) {
    x++;
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

volatile int x;
void __interrupt(2) incIsr(void) {
    x++;
}
```

Further Information
See the **Interrupts** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on writing interrupt functions.

4.30 weak Pragma

The IAR `weak` pragma makes the definition of a function or variable with external linkage a weak definition, alternatively, it creates a weak alias for another function or variable.

Suggested Replacement
There is no equivalent MPLAB XC8 pragma, but there is an attribute that performs a similar task.

Use the `weak` function attribute to have the declaration emitted as a weak symbol.

Caveats
There is no means of creating a weak alias for another function or variable.

Examples
Consider migrating IAR code such as:

```
#pragma weak foo
int foo;
```

to MPLAB XC8 code similar to:

```
extern int __attribute__((weak))foo;
```

Further Information
See the **Function Attributes** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on writing interrupt functions.

5. IAR Intrinsic Functions

This section shows a summary of IAR intrinsic functions and the best MPLAB XC8 replacement where that is available. These intrinsic functions and their MPLAB XC8 equivalents are discussed in more detail in the sections that follow. Note that what is referred to as an intrinsic function by IAR documentation is known as a built-in function in the MPLAB XC context. It is recommended that you compare the relevant sections in the *MPLAB® XC8 C Compiler User's Guide for AVR MCUs* with those in your *IAR C/C++ Compiler User Guide* to ensure that migrated code will work as expected in all situations.

Table 5-1. Summary of Suggested Migrations/Migration Suggestions from IAR Intrinsic Functions

IAR Intrinsic Function (links to explanatory section)	Suggested MPLAB XC8 Migration
__delay_cycles	Use the _delay_ms() or _delay_us() functions
__DES_decryption	No simple migration recommended
__DES_encryption	No simple migration recommended
__disable_interrupt	Use di() macro.
__enable_interrupt	Use the ei() macro.
__extended_load_program_memory	Use the address cast to a pointer and deferenced
__fractional_multiply_signed	Use in-line assembly to insert the fmuls instruction
__fractional_multiply_signed_with_unsigned	Use in-line assembly to insert the fmulsu instruction
__fractional_multiply_unsigned	Use in-line assembly to insert the fmul instruction
__get_interrupt_state	Use in-line assembly to directly fetch the state of the register
__indirect_jump_to	Use in-line assembly to insert the ijmp or eijmp instruction
__insert_opcode	Use in-line assembly to insert a .word directive
__lac	Use in-line assembly to insert the lac instruction
__las	Use in-line assembly to insert the las instruction
__lat	Use in-line assembly to insert the lat instruction
__load_program_memory	The address cast and dereferenced in C code
__multiply_signed	Use in-line assembly to insert the muls instruction
__multiply_signed_with_unsigned	Use in-line assembly to insert the mulsu instruction
__multiply_unsigned	Use in-line assembly to insert the mul instruction
__no_operation	Use the _NOP() macro.
__require	No simple migration recommended
__restore_interrupt	Use plain C code to write the SREG register
__reverse	No simple migration recommended
__save_interrupt	Plain C code to copy the SREG register
__set_interrupt_state	Use in-line assembly to directly set the state of the register
__sleep	Use in-line assembly to insert the sleep instruction
__swap_nibbles	No simple migration recommended

.....continued	
IAR Intrinsic Function (links to explanatory section)	**Suggested MPLAB XC8 Migration**
__watchdog_reset	Use in-line assembly to insert the `wdr` instruction
__xch	Use in-line assembly to insert the `xch` instruction

5.1 __delay_cycles Intrinsic Function

The IAR `__delay_cycles` intrinsic function adds code which delays execution by the specified constant number of cycles.

Suggested Replacement
There is no equivalent built-in function, but there are library routines that perform a similar task.

Use either the `_delay_ms()` or `_delay_us()` functions, specifying the delay time as a `double` argument.

Caveats
The macro `F_CPU` should be defined as a constant that specifies the CPU clock frequency (in Hertz). The compiler optimizers must be enabled for accurate delay times.

Examples
Consider migrating IAR code such as:

```
__delay_cycles(100);
```

to MPLAB XC8 code similar to:

```
#define F_CPU 1000000UL
_delay_us(100);
```

Further Information
See the **Library Functions** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on these functions.

5.2 __DES_decryption Intrinsic Function

The IAR `__DES_decryption` intrinsic function decrypts according to the Digital Encryption Standard (DES) of data against a key and returns the decrypted data.

Suggested Replacement
None.

5.3 __DES_encryption Intrinsic Function

The IAR `__DES_encryption` intrinsic function encrypts according to the Digital Encryption Standard (DES) of data against a key and returns the encrypted data.

Suggested Replacement
None.

5.4 __disable_interrupt Intrinsic Function

The IAR `__disable_interrupt` intrinsic function disables interrupts with the relevant instructions.

Suggested Replacement
There is an MPLAB XC8 macro that performs a similar task.

Use the `di()` macro to disable interrupts.

Caveats
The Common C Interface must be enabled using `-mext=cci` to use the macro replacement.

Examples
Consider migrating IAR code such as:

```
__disable_interrupt();
```

to MPLAB XC8 code similar to:

```
di();
```

and build with the `-mext=cci` option.

Further Information
See the **Enabling Interrupts** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on this macro.

5.5 __enable_interrupt Intrinsic Function

The IAR `__enable_interrupt` intrinsic function enables interrupts with the relevant instructions.

Suggested Replacement
There is an MPLAB XC8 macro that performs a similar task.

Use the `ei()` macro to enable interrupts.

Caveats
The Common C Interface must be enabled using `-mext=cci` to use the macro replacement.

Examples
Consider migrating IAR code such as:

```
__enable_interrupt();
```

to MPLAB XC8 code similar to:

```
ei();
```

and build with the `-mext=cci` option.

Further Information
See the **Enabling Interrupts** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on this macro.

5.6 __extended_load_program_memory Intrinsic Function

The IAR `__extended_load_program_memory` intrinsic function returns one byte from program memory.

Suggested Replacement
There is no equivalent MPLAB XC8 built-in function, but standard C code using an MPLAB XC8 qualifier can perform a similar task.

Cast the address to a `const __memx unsigned char *` type and dereference it in the usual way.

Caveats
None

Examples
Consider migrating IAR code such as:

```
volatile char x;
void foo(void) {
    x = __extended_load_program_memory(0x10000);
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

volatile char x;
void foo(void) {
    x = *((const __memx char *)0x10000);
}
```

Further Information
See the **Special Type Qualifiers** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on this qualifier.

5.7 __fractional_multiply_signed Intrinsic Function

The IAR `__fractional_multiply_signed` intrinsic function generates an `fmuls` instruction acting on the two arguments.

Suggested Replacement
There is no MPLAB XC8 equivalent built-in function; however, the instruction can be inserted explicitly using in-line assembly code.

Use in-line assembly to directly write an `fmuls` instruction.

Caveats
None

Examples
Consider migrating IAR code such as:

```
char x, y;
int z;
void foo(void) {
    z = __fractional_multiply_signed(x, y);
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

int avr_fmuls(char x, char y)
{
  int z;
  __asm__("fmuls %1, %2      \n\t"
          "movw %0, r0       \n\t"
          "clr __zero_reg__  \n\t"
          : "=r" (z)
          : "a" (x), "a" (y));
  return z;
}

char x, y;
int z;
void foo(void) {
    z = avr_fmuls(x, y);
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on adding in-line assembly.

5.8 __fractional_multiply_signed_with_unsigned Intrinsic Function

The IAR `__fractional_multiply_signed_with_unsigned` intrinsic function generates an `fmulsu` instruction acting on the two arguments.

Suggested Replacement
There is no MPLAB XC8 equivalent built-in function; however, the instruction can be inserted explicitly using in-line assembly code.

Use in-line assembly to directly write an `fmulsu` instruction.

Caveats
None

Examples
Consider migrating IAR code such as:

```
signed char x = 0x10;
unsigned char y = 0x20;
int z;
void foo(void) {
    z = __fractional_multiply_signed_with_unsigned(x, y);
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

int avr_fmulsu(char x, unsigned char y)
{
  int z;
  __asm__("fmulsu %1, %2      \n\t"
          "movw %0, r0       \n\t"
          "clr __zero_reg__  \n\t"
          : "=r" (z)
          : "a" (x), "a" (y));
  return z;
```

```
}

signed char x = 0x10;
unsigned char y = 0x20;
int z;
void foo(void) {
    z = avr_fmulsu(x, y);
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on adding in-line assembly.

5.9 __fractional_multiply_unsigned Intrinsic Function

The IAR `__fractional_multiply_unsigned` intrinsic function generates an `fmul` instruction acting on the two arguments.

Suggested Replacement
There is no MPLAB XC8 equivalent built-in function; however, the instruction can be inserted explicitly using in-line assembly code.

Use in-line assembly to directly write an `fmul` instruction.

Caveats
None

Examples
Consider migrating IAR code such as:

```
char x, y;
int z;
void foo(void) {
    z = __fractional_multiply_unsigned(x, y);
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

unsigned int avr_fmul(unsigned char x, unsigned char y)
{
  unsigned int z;
  __asm__("fmul %1, %2      \n\t"
          "movw %0, r0       \n\t"
          "clr __zero_reg__  \n\t"
          : "=r" (z)
          : "a" (x), "a" (y));
  return z;
}

char x, y;
int z;
void foo(void) {
    z = avr_fmul(x, y);
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on adding in-line assembly.

5.10 __get_interrupt_state Intrinsic Function

The IAR `__get_interrupt_state` intrinsic function returns the global interrupt state, which can be saved and used later by the `__set_interrupt_state` intrinsic function to restore the global interrupt state.

Suggested Replacement
There is no MPLAB XC8 equivalent built-in function; however, a similar code sequence can be inserted explicitly using in-line assembly code.

Use in-line assembly to directly fetch the state of the register, storing it in an appropriate C object.

Caveats
None

Examples
Consider migrating IAR code such as:

```
extern int mode;

void clearMode_safe()
{
  __istate_t s = __get_interrupt_state();
  __disable_interrupt();
  mode = 0x0;
  __set_interrupt_state(s);
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

unsigned char get_interrupt_state(void) {
  unsigned char s;
  __asm__ ("in %0, __SREG__"
           :"=d" (s));
  return s;
}

void set_interrupt_state(unsigned char s) {
  __asm__ ("sbrs %0, 7 \n\t"
           "rjmp .+4 \n\t"
           "sei \n\t"
           "rjmp .+2 \n\t"
           "cli \n\t"
           : : "r" (s));
}

extern int mode;

void clearMode_safe()
{
  unsigned char s = get_interrupt_state();
  di();
  mode = 0x0;
  set_interrupt_state(s);
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on adding in-line assembly.

5.11 __indirect_jump_to Intrinsic Function

The IAR `__indirect_jump_to` intrinsic function jumps to the specified address via a `ijmp` or `eijmp` instruction.

Suggested Replacement
There is no MPLAB XC equivalent built-in function; however, these instructions can be inserted explicitly using in-line assembly code.

Use in-line assembly to directly write an `ijmp` or `eijmp` instruction.

Caveats
Where an `eijmp` instruction has been inserted, make sure to restore the state of the EIND register afterward.

Examples
Consider migrating IAR code such as:

```
void leap(unsigned long addr) {
    __indirect_jump_to(addr);
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

unsigned char indirect_jump(unsigned long addr) {
#ifdef __AVR_HAVE_EIJMP_EICALL__
  /* Extract 3rd byte of address to h */
  unsigned char h = addr >> 16;
  /* Record existing EIND reg val in res */
  unsigned char res;
    __asm__("in %[EIND_SAVED], %[EIND_REG] \n\t"
            "out %[EIND_REG], %[HH] \n\t"
            "movw r30, %[ADDR] \n\t"
            "eijmp \n\t"
            : [EIND_SAVED] "=&r" (res)
            : [HH] "r" (h),
              [ADDR] "r" ((unsigned int)addr),
              [EIND_REG] "I" (_SFR_IO_ADDR(EIND))
            : "r30","r31");
  return res;
#else
    __asm__("movw r30, %0 \n\t"
            "ijmp \n\t"
            : : "r" ((unsigned int)addr) : "r30","r31");
   return 0;
#endif
}

void leap(unsigned long addr) {
    indirect_jump(addr);
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on this qualifier.

5.12 __insert_opcode Intrinsic Function

The IAR `__insert_opcode` intrinsic function inserts a `DW unsigned` directive.

Suggested Replacement
There is no MPLAB XC equivalent built-in function; however, instructions can be inserted explicitly using in-line assembly code.

Use in-line assembly to directly write a `.word` directive.

Caveats
None

Examples
Consider migrating IAR code such as:

```
char x, y;
int z;
void foo(void) {
    z = __insert_opcode(0x80a8);
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

char x, y;
int z;
void foo(void) {
    z = asm(".word 0x80a8");
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on adding in-line assembly.

5.13 __lac Intrinsic Function

The IAR `__lac` intrinsic function provides access to the `lac` instruction, available on AVRxm devices. The `lac` (Load And Clear) instruction loads the memory contents at the address held by the Z register into the Rd register specified while simultaneously clearing those bits at that same address held by the Z register that were set in the Rd register.

Suggested Replacement
There is no MPLAB XC8 equivalent built-in function; however, the instruction can be inserted explicitly using in-line assembly code.

Use in-line assembly to directly write an `lac` instruction.

Caveats
None

Examples
Consider migrating IAR code such as:

```
unsigned char loadAndClear(unsigned char regD, unsigned char * Zptr) {
    unsigned char cleared = __lac(regD, Zptr);
    return cleared;
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>
```

```
unsigned char lac(unsigned char v, unsigned char * addr) {
        __asm__("movw r30, %[ADDR] \n\t"
                "lac Z, %[Rd] \n\t"
                : [Rd] "+r" (v), "+m" (*addr)
                : [ADDR] "r" ((unsigned int)addr)
                : "r30", "r31");
        return v;
}

unsigned char loadAndClear(unsigned char regD, unsigned char * Zptr) {
    unsigned char cleared = lac(regD, Zptr);
    return cleared;
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on adding in-line assembly.

5.14 __las Intrinsic Function

The IAR `__las` intrinsic function provides access to the `las` (Load And Set) instruction, available on AVRxm devices. The `las` (Load And Set) instruction loads the memory contents at the address held by the Z register into the Rd register specified while simultaneously setting those bits at that same address held by the Z register that were set in the Rd register.

Suggested Replacement
There is no MPLAB XC8 equivalent built-in function; however, the instruction can be inserted explicitly using in-line assembly code.

Use in-line assembly to directly write an `las` instruction.

Caveats
None

Examples
Consider migrating IAR code such as:

```
unsigned char loadAndSet(unsigned char regD, unsigned char * Zptr) {
    unsigned char set = __las(regD, Zptr);
    return set;
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

unsigned char las(unsigned char v, unsigned char * addr) {
      __asm__("movw r30, %[ADDR] \n\t"
              "las Z, %[Rd] \n\t"
              : [Rd] "+r" (v), "+m" (*addr)
              : [ADDR] "r" ((unsigned int)addr)
              : "r30", "r31");
      return v;
    }

unsigned char loadAndSet(unsigned char regD, unsigned char * Zptr) {
    unsigned char set = las(regD, Zptr);
    return set;
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on adding in-line assembly.

5.15 __lat Intrinsic Function

The IAR `__lat` intrinsic function provides access to the `lat` instruction, available on AVRxm devices. The `lat` (Load And Toggle) instruction loads the memory contents at the address held by the Z register into the Rd register specified while simultaneously toggling those bits at that same address held by the Z register that were set in the Rd register.

Suggested Replacement
There is no MPLAB XC8 equivalent built-in function; however, the instruction can be inserted explicitly using in-line assembly code.

Use in-line assembly to directly write an `lat` instruction.

Caveats
None

Examples
Consider migrating IAR code such as:

```
unsigned char loadAndToggle(unsigned char regD, unsigned char * Zptr) {
    unsigned char toggled = __lat(regD, Zptr);
    return toggled;
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

unsigned char lat(unsigned char v, unsigned char * addr) {
    __asm__("movw r30, %[ADDR] \n\t"
            "lat Z, %[Rd] \n\t"
            : [Rd] "+r" (v), "+m" (*addr)
            : [ADDR] "r" ((unsigned int)addr)
            : "r30", "r31");
    return v;
}

unsigned char loadAndClear(unsigned char regD, unsigned char * Zptr) {
    unsigned char toggled = lat(regD, Zptr);
    return toggled;
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on adding in-line assembly.

5.16 __load_program_memory Intrinsic Function

The IAR `__load_program_memory` intrinsic function reads one byte from the specified code memory within the lower 64kB of program memory.

Suggested Replacement
There is no MPLAB XC8 equivalent built-in function; however, library functions and plain C code can perform a similar task.

Use the `pgm_read_byte()` function to read a byte of program memory.

Alternatively, cast the address to a `const __flash unsigned char *` pointer and dereference it in the usual way.

Caveats
The `pgm_read_byte()` function takes an integer argument, not a pointer.

Examples
Consider migrating IAR code such as:

```
volatile unsigned char val;
__flash unsigned char myVar = 0x55;

void foo(void) {
  unsigned char __flash * dp = &myVar;
  val = __load_program_memory(dp);
}
```

to MPLAB XC8 code similar to:

```
#include <avr/pgmspace.h>
const __flash myVar = 0x55;

int main(void)
{
  unsigned char val;
  val = pgm_read_byte(&myVar);
}
```

Further Information
See the **Library Functions** and **Special Type Qualifiers** sections in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on the library function and qualifier.

5.17 __multiply_signed Intrinsic Function

The IAR `__multiply_signed` intrinsic function provides access to the `muls` instruction.

Suggested Replacement
There is no MPLAB XC8 equivalent built-in function; however, the instruction can be inserted explicitly using in-line assembly code.

Use in-line assembly to directly write an `muls` instruction.

Caveats
None

Examples
Consider migrating IAR code such as:

```
int operation(signed char x, signed char y)
{
    int result = __multiply_signed(x, y);
    return result;
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

int multiply_signed(signed char x, signed char y)
{
```

```
  int z;
  __asm__("muls %1, %2      \n\t"
          "movw %0, r0       \n\t"
          "clr __zero_reg__  \n\t"
          : "=r" (z)
          : "a" (x), "a" (y));
  return z;
}

int operation(signed char x, signed char y)
{
    int result = multiply_signed(x, y);
    return result;
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on adding in-line assembly.

5.18 __multiply_signed_with_unsigned Intrinsic Function

The IAR `__multiply_signed_with_unsigned` intrinsic function provides access to the `mulsu` instruction.

Suggested Replacement
There is no MPLAB XC8 equivalent built-in function; however, the instruction can be inserted explicitly using in-line assembly code.

Use in-line assembly to directly write an `mulsu` instruction.

Caveats
None

Examples
Consider migrating IAR code such as:

```
int operation(signed char x, unsigned char y)
{
    int result = __multiply_signed_with_unsigned(x, y);
    return result;
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

int multiply_signed_with_unsigned(signed char x, unsigned char y)
{
  int z;
  __asm__("mulsu %1, %2      \n\t"
          "movw %0, r0       \n\t"
          "clr __zero_reg__  \n\t"
          : "=r" (z)
          : "a" (x), "a" (y));
  return z;
}

int operation(signed char x, unsigned char y)
{
    int result = multiply_signed_with_unsigned(x, y);
    return result;
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on adding in-line assembly.

5.19 __multiply_unsigned Intrinsic Function

The IAR `__multiply_unsigned` intrinsic function provides access to the `mul` instruction.

Suggested Replacement
There is no MPLAB XC8 equivalent built-in function; however, the instruction can be inserted explicitly using in-line assembly code.

Use in-line assembly to directly write an `mul` instruction.

Caveats
None

Examples
Consider migrating IAR code such as:

```
unsigned int operation(unsigned char x, unsigned char y)
{
    unsigned int result = __multiply_unsigned(x, y);
    return result;
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

unsigned int multiply_unsigned(unsigned char x, unsigned char y)
{
  unsigned int z;
  __asm__("mul %1, %2      \n\t"
          "movw %0, r0       \n\t"
          "clr __zero_reg__  \n\t"
          : "=r" (z)
          : "r" (x), "r" (y));
  return z;
}

unsigned int operation(unsigned char x, unsigned char y)
{
    unsigned int result = multiply_unsigned(x, y);
    return result;
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on adding in-line assembly.

5.20 __no_operation Intrinsic Function

The IAR `__no_operation` intrinsic function inserts a `nop` instruction.

Suggested Replacement
There is an MPLAB XC8 macro that performs a similar task.

Use the `_NOP()` macro.

Caveats
None

Examples
Consider migrating IAR code such as:

```
__no_operation();
```

to MPLAB XC8 code similar to:

```
_NOP();
```

Further Information
See the **Library Functions** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on this macro.

5.21 __require Intrinsic Function

The IAR `__require` intrinsic function ensures a symbol is output, even if it appears to be unreferenced..

Suggested Replacement
None.

5.22 __restore_interrupt Intrinsic Function

The IAR `__restore_interrupt` intrinsic function restores the interrupt flag to a state obtained using the `__save_interrupt` intrinsic function.

Suggested Replacement
There is no equivalent MPLAB XC8 built-in function, but the SREG registers can be updated with plain C code.

Use code that performs a bitwise OR of the SREG register and a saved value obtained from that same register before the interrupt flags were potentially changed.

Caveats
None

Examples
Consider migrating IAR code such as:

```
unsigned char saved;

void foo(void) {
    saved = __save_interrupt();
    __disable_interrupt();
    /* Critical section goes here */
    __restore_interrupt(saved);
}
```

to MPLAB XC8 code similar to:

```
volatile unsigned char saved;

void foo(void) {
    saved = (SREG & 0x80);
    di();
    /* Critical section goes here */
```

```
    SREG |= saved;
}
```

5.23 __reverse Intrinsic Function

The IAR `__reverse` intrinsic function returns its parameter with its byte order reversed.

Suggested Replacement
None.

5.24 __save_interrupt Intrinsic Function

The IAR `__save_interrupt` intrinsic function saves the interrupt flag so that it may later be used by the `__restore_interrupt` intrinsic function.

Suggested Replacement
There is no equivalent MPLAB XC8 built-in function, but the SREG registers can be updated with plain C code.

Use code that copies the interrupt bit within the SREG register into an ordinary variable.

Caveats
None

Examples
Consider migrating IAR code such as:

```
unsigned char saved;

void foo(void) {
    saved = __save_interrupt();
    __disable_interrupt();
    /* Critical section goes here */
    __restore_interrupt(saved);
}
```

to MPLAB XC8 code similar to:

```
volatile unsigned char saved;

void foo(void) {
    saved = (SREG & 0x80);
    di();
    /* Critical section goes here */
    SREG |= saved;
}
```

5.25 __set_interrupt_state Intrinsic Function

The IAR `__set_interrupt_state` intrinsic function sets the global interrupt state to that saved by a previous call to the `__get_interrupt_state` intrinsic function.

Suggested Replacement
There is no MPLAB XC8 equivalent built-in function; however, a similar code sequence can be inserted explicitly using in-line assembly code.

Use in-line assembly to directly set the state of the register to that saved by previously called in-line assembly code.

Caveats
None

Examples
Consider migrating IAR code such as:

```
extern int mode;

void clearMode_safe()
{
  __istate_t s = __get_interrupt_state();
  __disable_interrupt();
  mode = 0x0;
  __set_interrupt_state(s);
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

unsigned char get_interrupt_state(void) {
  unsigned char s;
  __asm__ ("in %0, __SREG__"
           :"=d" (s));
  return s;
}

void set_interrupt_state(unsigned char s) {
  __asm__ ("sbrs %0, 7 \n\t"
           "rjmp .+4 \n\t"
           "sei \n\t"
           "rjmp .+2 \n\t"
           "cli \n\t"
           : : "r" (s));
}

extern int mode;

void clearMode_safe()
{
  unsigned char s = get_interrupt_state();
  di();
  mode = 0x0;
  set_interrupt_state(s);
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on adding in-line assembly.

5.26 __sleep Intrinsic Function

The IAR `__sleep` intrinsic function inserts a `sleep` instruction.

Suggested Replacement
There is no equivalent MPLAB XC8 built-in function, but a `sleep` instruction can be inserted using in-line assembly.

Use `asm("sleep");` to insert the `sleep` instruction into the generated code.

Caveats
None

Examples
Consider migrating IAR code such as:

```
void waitForService(void)
{
    __enable_interrupt();
    __sleep();
}
```

to MPLAB XC8 code similar to:

```
void waitForService(void)
{
    ei();
    asm("sleep");
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on in-line assembly and this built-in function.

5.27 __swap_nibbles Intrinsic Function

The IAR `__swap_nibbles` intrinsic function returns the argument with the upper and lower nibbles swapped.

Suggested Replacement
None.

5.28 __watchdog_reset Intrinsic Function

The IAR `__watchdog_reset` intrinsic function inserts a `wdr` (watchdog reset) instruction.

Suggested Replacement
There is no equivalent MPLAB XC8 built-in function, but a `wdr` instruction can be inserted using in-line assembly.

Use `asm("wdr");` to insert the `wdr` instruction into the generated code..

Caveats
None

Examples
Consider migrating IAR code such as:

```
void main(void)
{
    __watchdog_reset();
}
```

to MPLAB XC8 code similar to:

```
void main(void)
{
    asm("wdr");

}
```

Further Information
See the **In-line Assembly** and **???** and section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on in-line assembly.

5.29 __xch Intrinsic Function

The IAR `__xch` intrinsic function provides access to the `xch` instruction, available on AVRxm devices. The `xch` (Exchange) instruction swaps the memory contents at the address held by the Z register with the Rd register specified.

Suggested Replacement
There is no MPLAB XC equivalent built-in function; however, the instruction can be inserted explicitly using in-line assembly code.

Use in-line assembly to directly write an `xch` instruction.

Caveats
None

Examples
Consider migrating IAR code such as:

```
unsigned char exchange(unsigned char regD, unsigned char * Zptr) {
    unsigned char swapped = __xch(regD, Zptr);
    return swapped;
}
```

to MPLAB XC8 code similar to:

```
#include <xc.h>

unsigned char xch(unsigned char v, unsigned char* addr) {
  __asm__("movw r30, %[ADDR] \n\t"
          "xch Z, %[Rd] \n\t"
          : [Rd] "+r" (v), "+m" (*addr)
          : [ADDR] "r" ((unsigned int)addr)
          : "r30", "r31");
  return v;
}

unsigned char exchange(unsigned char regD, unsigned char * Zptr) {
    unsigned char swapped = xch(regD, Zptr);
    return swapped;
}
```

Further Information
See the **In-line Assembly** section in the *MPLAB XC8 C Compiler User's Guide for AVR MCUs* for more information on adding in-line assembly.

6. Document Revision History

Revision A (July 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: 978-1-6683-0863-9

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