M16C/R8C IAR Embedded Workbench

Migration Guide

for Renesas

M16C/1X–3X, 6X, and R8C Series of CPU Cores

EDITION NOTICE

First edition: October 2004

Part number: MM16C-1

This guide applies to the M16C/R8C IAR Embedded Workbench™ version 3.x

Contents

Tables

Migrating from version 2.x to version 3.x

This guide presents the major differences between version 2.x and version 3.x of M16C/R8C IAR Embedded Workbench, and describes the migration considerations. Hereafter, the two versions are referred to as version 2.x and version 3.x, respectively.

Note that if you are migrating from M16C/R8C IAR Embedded Workbench version 1.x, you must first read the chapter *Migrating from version 1.x to version 2.x*.

Migration considerations

To migrate your old project consider the following:

- Changes in IAR Embedded Workbench, see *IAR Embedded Workbench IDE*, page 1
- Changes in the runtime environment, see *Runtime library and object files*, page 3
- Location of functions declared `__tiny_func`. In version 3.x, functions declared `__tiny_func` will be automatically located in the `TINYFUNC` segment, which should be placed in the special page area. In version 2.x, you have to manually make sure that these functions are located in a specific segment, which is to be placed within the appropriate address range.

Note that not all items in the migration procedure may be relevant for your project. Consider carefully what actions are needed in your case.

IAR Embedded Workbench IDE

Version 3.x provides new improved project management with support for complex project setup. The former *project window*—which could manage one separate project—has been exchanged with a *workspace window*. This workspace window can manage several projects and multiple build configurations for each project. For more information about project management in version 3.x, see the *IAR Embedded Workbench™ IDE User Guide.*

Upgrading to the new version of the IAR Embedded Workbench IDE should still be a smooth process, but you should consider the following:

- Project file and project setup
- Project options
- C-SPY layout files.

PROJECT FILE AND PROJECT SETUP

If you are using the IAR Embedded Workbench IDE, follow these steps to verify that your project file has been properly converted:

1. Start your new version of M16C/R8C IAR Embedded Workbench and create a new workspace by choosing **File>New** and then **Workspace**.
2. Choose **Project>Add Existing Project** to insert your old project into the workspace. This step will create two new project files with the same name as the old file, but with the extensions `ewp` and `ewd`. The `ewp` file contains all settings required to build the application, while the `ewd` file contains all settings related to the debugger. The old project file will remain untouched.
3. Verify that your options have been set up correctly.

   To generate a text file with the command line equivalents of the project options in your old project, see *Migrating project options*, page 2.

   Also, set any new options.
4. If you have your own linker command file, compare this file with the original file in the old installation and make the required changes in a copy of the corresponding file in the new installation.

MIGRATING PROJECT OPTIONS

Since the available compiler options differ between version 1.x and version 2.x, you should verify your option settings after you have converted an old project.

If you are using the command line interface, you can simply compare your makefile with the option tables in this section, and modify the makefile accordingly.

If you are using the IAR Embedded Workbench IDE, all option settings are automatically converted during the project conversion.

However, it is still recommended to verify the options manually. Follow these steps:

1. Open the old project in the old IAR Embedded Workbench version.
2. In the project window, select the project level to get information about options on all levels in your project.

3 To save the project settings to a file, right-click in the project window. On the context menu that appears, choose **Save As Text**, and save the settings to an appropriate destination.

4 Use this file and the option tables in this section to verify whether the options you used in your old project are still available or needed. Also check whether you need to use any of the new options.

For information about where to set the equivalent options in the IAR Embedded Workbench IDE, see the *M16C/R8C IAR C/C++ Compiler Reference Guide*.

A new compiler optimization, **Type-based alias analysis**, is enabled by default. This optimization can be disabled with the option `--no_tbaa` or by deselecting the IAR Embedded Workbench counterpart.

C-SPY LAYOUT FILES

Due to a new improved window management system, the C-SPY layout files support in version 2.x has been removed. Any custom-made `lew` files can be safely removed from your projects.

Runtime library and object files

In version 3.x, two sets of runtime libraries are provided—CLIB and DLIB. CLIB can be used in the same way as before. DLIB has been extended with new possibilities for configuration.

To build code produced by version 3.x of the compiler, you must use the runtime environment components it provides. It is not possible to link object code produced using version 3.x with components provided with version 2.x.

For information about how to migrate from CLIB to DLIB, see *Migrating from CLIB to DLIB*, page 6. For more information about the two libraries, and the runtime environment they provide see the *IAR Runtime Environment and Library Guide*.

COMPILING AND LINKING WITH THE DLIB RUNTIME LIBRARY

In earlier versions, the choice of runtime library did not have any impact on the compilation. In M16C/R8C IAR Embedded Workbench version 3.x, this has changed. Now you can configure the runtime library to contain the features that are needed by your application.

One example is input and output. An application may use the `fprintf` function for terminal I/O (`stdout`), but the application does not use file I/O functionality on file descriptors associated with the files. In this case the library can be configured so that code related to file I/O is removed but still provides terminal I/O functionality.

This configuration involves the library header files, for example `stdio.h`. This means that when you build your application, the same header file setup must be used as when the library was built. The library setup is specified in a *library configuration file*, which is a header file that defines the library functionality.

When building an application using the IAR Embedded Workbench, there are three library configuration alternatives to choose between: **Normal**, **Full**, and **Custom**. **Normal** and **Full** are prebuilt library configurations delivered with the product, where **Normal** should be used in the above example with file I/O. **Custom** is used for custom-built libraries. Note that the choice of the library configuration file is handled automatically.

When building an application from the command line, you must use the same library configuration file as when the library was built. For the prebuilt libraries (`r34`) there is a corresponding library configuration file (`h`), which has the same name as the library. The files are located in the `m16c\lib` directory. The command lines for specifying the library configuration file and library object file could look like this:

```
iccm16c --DLIB_config ...\m16c\lib\dlib dlm16cffffwc.h
xlink dlm16cffffwc.r34
```

In case you intend to build your own library version, use the default library configuration file `dlm16cCustom.h`.

To take advantage of the new features it is recommended that you read about the runtime environment in the *IAR Runtime Environment and Library Guide*.

DYNAMIC MEMORY ALLOCATION—HEAP

In version 3.x, dynamic memory allocation is possible in the memories data13, data20, and far. To access a heap in a specific memory, use the appropriate memory attribute as a prefix to the standard functions `malloc`, `free`, `calloc`, and so forth. For example:

```
__data16_malloc
```

Each heap will reside in a segment with the name `_HEAP` prefixed by a memory attribute, for example:

```
DATA16_HEAP
```

Read more about this in the *IAR Linker and Library Tools Reference Guide.*

PROGRAM ENTRY

By default, the linker includes all `root` declared segment parts in program modules when building an application. However, there is a new mechanism that affects the load procedure.

There is a new linker option **Entry label** (`-s`) to specify a *start label*. By specifying the start label, the linker will look in all modules for a matching start label, and start loading from that point. Like before, any program modules containing a root segment part will also be loaded.

In version 3.x, the default program entry label in `cstartup.s34` is `__program_start`, which means the linker will start loading from there. The advantage of this new behavior is that it is much easier to override `cstartup.s34`.

If you build your application in the IAR Embedded Workbench, just add your customized `cstartup` file to your project. It will then be used instead of the `cstartup` module in the library. It is also possible to switch startup files just by overriding the name of the program entry point.

If you build your application from the command line, the `-s` option must be explicitly specified when linking a C/C++ application. If you link without the option, the resulting output executable will be empty because no modules will be referred to.

SYSTEM INITIALIZATION—CSTARTUP

The content of the `cstartup.s34` file has been split up into three files:

```
cstartup.s34, cmain.s34, cexit.s34
```

Now, the `cstartup.s34` file only contains exception vectors and initial startup code to setup stacks and processor mode. Note that the `cstartup.s34` file is the only one of these three files that may require any modifications.

The `cmain.s34` file initializes data segments and executes C++ constructors. The `cexit.s34` file contains termination code, for example, execution of C++ destructors.

For applications that use a modified copy of `cstartup.s34`, you must adapt it to the new file structure.

MIGRATING FROM CLIB TO DLIB

There are some considerations to have in mind when if you want to migrate from the CLIB, the legacy C library, to the modern DLIB C/C++ library:

- The CLIB `exp10()` function defined in `iccext.h` is not available in DLIB.
- The DLIB library uses the low-level I/O routines `__write` and `__read` instead of `putchar` and `getchar`.
- If the heap size in your version 2.x project using CLIB was defined in a file named `heap.c`, you must now set the heap size either in the extended linker command file (`*.xcl`) or in the Embedded Workbench to use the DLIB library.

You should also see the chapter *The DLIB runtime environment* in the *IAR Runtime Environment and Library Guide*.

Migrating from version 1.x to version 2.x

This chapter contains information about migrating from version 1.x to version 2.x of M16C IAR Embedded Workbench. Follow the instructions in this chapter first if you are migrating from version 1.x and then read the chapter *Migrating from version 2.x to version 3.x*, which contains information about migrating projects from M16C/R8C IAR Embedded Workbench version 2.x to 3.x.

Differences

The major differences between version 1.x and 2.x are:

The most obvious difference between version 1.x and 2.x is that with version 2.x, support for Embedded C++ is available.

Moreover, version 2.x adheres more strictly to the ISO/ANSI C standard; for example, it is possible to use pragma directives instead of extended keywords for defining special function registers (SFRs).

The checking of data types now adheres more strictly to the ISO/ANSI C standard, compared to version 1.x.

Note: It is important to be aware of the fact that code written for version 1.x may generate warnings or errors in version 2.x.

The migration process

To migrate your old project, follow the described migration process. Note that not all steps in the described migration process may be relevant for your project.

In short, to migrate from version 1.x to version 2.x, consider the following:

1. Replace version 1.x extended keywords in the source code with version 2.x keywords. For more information, see *Extended keywords*, page 12.
2. Replace version 1.x pragma directives with version 2.x directives. Notice that the behavior differs between the two versions; for more information, see *Pragma directives*, page 16.

3 Replace version 1.x intrinsic functions with version 2.x intrinsic functions, whenever needed. For more information, see *Intrinsic functions*, page 18.

4 Choose the appropriate version 2.x compiler options. For more information, see *Compiler options*, page 9.

5 Modify a copy of the supplied version 2.x linker command file template to suit your application requirements.

6 Link the code and run the project in the IAR C-SPY™ Debugger.

Note: A file `migration.h`, which contains translation macros that make the migration easier, is provided with the product. If you use this file, you can skip steps 1 and 3 above.

7 Make sure not to use nested comments in your source code. In version 2.x, nested comments are never allowed.

8 The version 2.x compiler uses a different C parser, and a large number of new optimizations have been added. Depending on your old source code, this might require you to modify your source code. One example of this is a simple delay loop, such as:

```
i = 50000;
do {i--;}
while (i-- != 0);
```

This code will be removed by the optimizer, unless you declare the variable `i` as `volatile`.

In order to produce more efficient code, the compiler performs transformations like, for example, removing redundant calculations, replacing division by shift, and removing useless calculations. Code that the compiler considers as *not useful* is removed; this may cause unexpected effects like in this example.

9 Version 2.x will by default not accept preprocessor expressions containing any of the following:

- Floating-point expressions
- Basic type names and `sizeof`
- All symbol names (including typedefs and variables).

With the option `--migration_preprocessor_extensions`, version 2.x will accept such non-standard expressions. For details about this option, see the *M16C/R8C IAR C/C++ Compiler Reference Guide*.

The remainder of this chapter describes the differences between version 1.x and 2.x in detail.

Compiler options

The command line options in version 2.x follow two different syntax styles:

- Long option names containing one or more words prefixed with two dashes, and sometimes followed by an equal sign and a modifier, for example `--strict_ansi` and `--module_name=test`.
- Short option names consisting of a single letter prefixed with a single dash, and sometimes followed by a modifier, for example `-r`.

Some options appear in one style only, while other options appear as synonyms in both styles. A number of new command line options have been added. For a complete list of the available command line options, see the *M16C/R8C IAR C/C++ Compiler Reference Guide*.

REMOVED OPTIONS

The following table shows the command line options that have been removed:

Old option	Description
`-C`	Nested comments
`-F`	Form-feed in list file after each function
`-G`	Opens standard input as source; replaced by - (dash) as source file name in version 2.x
`-g`	Global strict type checking; in version 2.x, global strict type checking is always enabled
`-gO`	No type information in object code
`-i`	Adds #include file text
`-K`	`'//'` comments; in version 2.x, `'//'` comments are allowed unless the option `--strict_ansi` is used
`-m[sSmMlLbB]`	Replaced by `--data_model`, `--constant_data`, and `--variable_data`
`-p`*nn*	Lines/page
`-T`	Active lines only
`-t`*n*	Tab spacing
`-U`*symb*	Undefined preprocessor symbol
`-X`	Explains C declarations
`-x[DFT2]`	Cross-reference

Table 1: Version 1.x compiler options not available in version 2.x

IDENTICAL OPTIONS

The following table shows the command line options that are *identical* in version 1.x and version 2.x:

Option	Comment
`-D`*`symb`*`=`*`value`*	Defines symbols
`-e`	Language extensions
`-f` *`filename`*	Extends the command line
`-h`	Enables debug code for HP debugger
`-I`	Defines include paths (The syntax is more free in version 2.x)
`-o` *`filename`*	Sets object filename
`-s[0–9]`	Optimizes for speed
`-u`	Data alignment
`-z[0–9]`	Optimizes for size
`-y`	Writable strings
`-2`	Treats doubles as 64-bit floating-point numbers.

Table 2: Compiler options identical in both compiler versions

RENAMED OR MODIFIED OPTIONS

The following version 1.x command line options have been *renamed* and/or *modified*:

Old option	New option	Description

`-A`
`-a` *`filename`*
	`-la .`
	`-la` *`filename`*
		Assembler output; see *Filenames*, page 11

`-b`	`--library_module`	Makes an object a library module
`-c`	`--char_is_signed`	'`char`' is 'signed `char`'
`-gA`	`--strict_ansi`	Flags old-style functions
`-H`*`name`*	`--module_name=`*`name`*	Sets object module name
`-L[`*`prefix`*`]`, `-l` *`filename`*	`-l[c|C|a|A][N][H]` *`filename`*	Generates list file; the modifiers specify the type of list file to create
`-N`*`prefix`*, `-n` *`filename`*	`--preprocess=[c][n][l]` *`filename`*	Preprocessor output

`-q`	`-lA .`
	`-lC .`
		Inserts mnemonics; list file syntax has changed

`-r[012][i][n]`	`-r` `--debug`	Generates debug information; the modifiers have been removed

Table 3: Renamed or modified options

Old option	New option	Description
`-S`	`--silent`	Sets silent operation
`-W{rs}`	`--workseg_area{=rs}`	Specifies the space reserved in the `saddr` area for the `WRKSEG` segment.

Table 3: Renamed or modified options (Continued)

Note: A number of new command line options have been added in version 2.x. For a complete list of the available command line options, see *M16C/R8C IAR C/C++ Compiler Reference Guide*.

FILENAMES

In version 1.x, file references can be made in either of the following ways:

- With a specific filename, and in some cases with a default extension added, using a command line option such as `-a` *`filename`* (assembler output to named file).
- With a prefix string added to the default name, using a command line option such as `-A[`*`prefix`*`]` (assembler output to prefixed filename).

In version 2.x, a file reference is always regarded as a *file path* that can be a directory which the compiler will check and then add a default filename to, or a *filename*.

The following table shows some examples where it is assumed that the source file is named `test.c`, `myfile` is *not* a directory, and `mydir` is a directory:

Old command	New command	Result
`-l myfile`	`-l myfile`	myfile.lst
`-Lmyfile`	`-l myfiletest`	myfiletest.lst
`-L`	`-l .`	test.lst
`-Lmydir/`	`-l mydir`	mydir/test.lst

Table 4: Specifying filename and directory in version 1.x and version 2.x

LIST FILES

In version 1.x, only one C list file and one assembler list file can be produced; in version 2.x there is no upper limit on the number of list files that can be generated. The new command line option `-l[c|C|a|A][N][H]` *`filename`* is used for specifying the behavior of each list file.

Extended keywords

The set of language extensions has changed in version 2.x. Some extensions have been added, some extensions have been removed, and for some of them the syntax has changed. There is also a rare case where an extension has a different interpretation if `typedefs` are used. This is described in the following section.

In version 2.x, all extended keywords except `asm` start with two underscores, for example `__data16`. For detailed information about the extended keywords, see the *M16C/R8C IAR C/C++ Compiler Reference Guide*.

STORAGE MODIFIERS

Both version 1.x and version 2.x allow keywords that specify memory location. Each of these keywords can be used either as a placement attribute for an object, or as a pointer type attribute denoting a pointer that can point to the specified memory.

When the keywords are used directly in the source code, they behave in a similar way in version 1.x and version 2.x. The usage of type definitions and extended keywords is, however, more strict in version 2.x than in version 1.x.

Usage in version 1.x

Products based on the previous compiler technology behave unexpectedly in some cases:

```
typedef int near NINT;
NINT a,b;
NINT huge c;      /* Illegal */
NINT *p;          /* p stored in near memory, points to
                     default memory attribute */
```

The first variable declaration works as expected, that is `a` and `b` are located in near memory.

The declaration of `c` is however illegal, except when near is the default memory, in which case there is no need for an extended keyword in the `typedef`.

In the last declaration, the `near` keyword of the `typedef` affects the location of the pointer variable `p`, not the pointer type. The pointer type is the default, which is given by the memory model.

Usage in version 2.x

The corresponding example for version 2.x is:

```
typedef int __data16 NINT;
NINT a,b;
NINT __data20 c;  /* c stored in data20 memory --
                     override attribute in typedef */
NINT *p;          /* p stored in default memory, points
                    to data16 memory */
```

The declarations of `c` and `p` differ. The `__data20` keyword in the declaration of `c` will always compile. It overrides the keyword of the `typedef`. In the last declaration the `__data16` keyword of the `typedef` affects the type of the pointer. It is thus a `__data16` pointer to `int`. The location of the variable `p` is however not affected.

CALLING CONVENTION

For backward compatibility, the M16C/R8C IAR C/C++ Compiler version 2.x also supports the calling convention used by version 1.x of the compiler. For information about the old calling conventions, see the user documentation provided with that compiler version. To use the old calling conventions, define and declare your functions with the `__simple` keyword, which is available for backward compatibility.

Regardless of using the `__simple` keyword, it will not be possible to link user object files (`r34`) created with the compiler version 1.x with object files created with version 2.x.

__NO_INIT

In version 2.x `__no_init` can be used, optionally together with a keyword, for suppressing initialization of a variable at system startup, for example:

```
__near __no_init char buffer [1000];
```

In version 1.x, this keyword was not available.

BIT VARIABLES

A bit variable in version 1.x is a volatile boolean variable that can have an absolute bit-address, be co-located with an SFR or be a relocatable object, like ordinary variables. For example:

```
bit a = 87;       /* at bit-address 87 (1.x) */
bit p0 = PORT.5;  /* bit 5 of port (1.x) */
bit r;            /* relocatable bit (1.x) */
```

Version 2.x uses bitfields of width 1 to implement bit variables. The extended language feature anonymous structs allows the bits, which are struct members, to be used as if they were variables in the enclosing scope. The keyword `bit` is not available in version 2.x. For additional information about anonymous structs, see the *M16C/R8C IAR C/C++ Compiler Reference Guide*.

The following example shows an anonymous struct in version 2.x:

```
/* anonymous struct */
struct {
  char b0:1, b1:1, b2:1, :5, b7:1;
};
char foo() { return b7; }
void bar() { b0 = 1; }
```

A relocatable bit variable `my_bit` can be declared in version 2.x using the `__bitvar` keyword. The variable will be stored in the `BITVARS` segment, where each variable only occupies one bit. For example:

```
__bitvar struct {unsigned char my_bit:1;}
```

For more information, see *__bitvar*, page 126.

To declare an absolute-located bit, the bit address must first be converted to a byte address. The bit `a`, in the above example, has bit address 87. Division by 8 yields byte address 10 and remainder 7, the latter of which is the bit-offset in that byte. Thus the corresponding version 2.x declaration is:

```
volatile __near struct { char :7, a:1; } @ 10;
```

Anonymous unions are used for locating an SFR and a bit field at the same address.

The declaration of `PORT` (address 100) and `p0` (bit 5 of `PORT`) are combined in the following way:

```
/* anonymous union */
volatile __near union {
char PORT;
struct { char :5, p0:1; };
} @ 100;
```

The version 2.x notation is not as brief as the one used in version 1.x. It is, on the other hand, more flexible. Bit fields can have any width (not only 1), can be located in any memory (not restricted to near) and are not necessarily volatile.

INTERRUPT FUNCTIONS AND VECTORS

In version 1.x, a vector offset can be attached to an `interrupt` function with the `#pragma function` directive or directly in the source code, for example:

```
interrupt [32] void f(void);
```

where `32` is the *vector offset* in the vector table. The vector offset is the vector number times the size of a vector entry, which could be 2 or 4 bytes depending on the memory model used.

In version 2.x a vector number can be attached to an `__interrupt` function only by using the `#pragma vector` directive, for example:

```
#pragma vector=8
__interrupt void f(void);
```

where `8` is the *vector number.*

USING

The `using` keyword is not available in version 2.x. Consider the following example from version 1.x:

```
interrupt [32]using [ALTERNATE_SET] void f(void);
```

In version 2.x, the corresponding definition is (using the `__regbank_interrupt` keyword):

```
#pragma vector = 8
__regbank_interrupt void f(void);
```

Note: `ALTERNATE_SET` was the only value accepted by version 1.x for the `using` keyword.

SFR AND ABSOLUTE LOCATED VARIABLES

In version 1.x, the `sfr` and `sfrp` keywords denote an object of byte or word size residing in the SFR (Special Function Register) memory area for the chip, and being of the `volatile` type. The SFR is always located at an absolute address, for example:

```
sfr PORT=100;
```

In version 2.x, `sfr` and `sfrp` are not available. Instead you can:

- Place any object into the SFR memory, or any other memory, by using a memory attribute.
- Locate any object at an absolute address by using the `#pragma location` directive or by using the locator operator `@`, for example:
  ```
  __no_init long PORT @ 100;
  ```

- Use the `volatile` attribute on any type, for example:
  `volatile __no_init char PORT@100;`

See the *M16C/R8C IAR C/C++ Compiler Reference Guide* for detailed information about the extended keywords.

Pragma directives

Version 1.x and version 2.x have different sets of pragma directives for specifying attributes, and they also behave differently:

- In version 1.x, `#pragma memory` specifies the default location of data objects, and `#pragma function` specifies the default location of functions. They change the default attribute to use for declared objects; they do not have an effect on pointer types.
- In version 2.x, the `#pragma type_attribute` and `#pragma object_attribute` directives only change the next declared object or `typedef`.

See the *M16C/R8C IAR C/C++ Compiler Reference Guide* for information about the pragma directives.

REMOVED DIRECTIVES

The following pragma directives have been removed:

- `alignment`
- `codeseg`
- `function`
- `memory`
- `warnings`

They are recognized and will give a diagnostic message but will not work in version 2.x.

Note: Instead of the `#pragma codeseg` directive, you can use the `#pragma location` directive or the `@` operator for specifying an absolute location.

MODIFIED DIRECTIVES

The following table shows the mapping of pragma directives:

Old directive	New pragma directive
`#pragma function=interrupt`	`#pragma type_attribute=__interrupt`
`#pragma function=monitor`	`#pragma type_attribute=__monitor`
`#pragma memory=constseg`	`#pragma constseg, #pragma location`

Table 5: Old and new pragma directives

Old directive	New pragma directive
`#pragma memory=dataseg`	`#pragma dataseg, #pragma location`
`#pragma memory=near`	`#pragma memory=__data16`

Table 5: Old and new pragma directives (Continued)

It is important to note that the new directives `#pragma type_attribute`, `#pragma object_attribute`, and `#pragma vector` affect only the *first* of the declarations that follow after the directive. In the following example, `x` is affected, but `z` and `y` are not affected by the directive:

```
#pragma object_attribute=__no_init
int x,z;
int y;
```

The version 2.x directives `#pragma constseg` and `#pragma dataseg` are active until they are explicitly turned off with the directive `#pragma constseg=default` and `#pragma dataseg=default`, respectively. For example:

```
#pragma constseg=myseg
__no_init f;
#pragma constseg=default
```

IDENTICAL DIRECTIVES

The following pragma directives are identical in version 1.x and version 2.x:

```
#pragma language=extended
#pragma language=default
```

NEW DIRECTIVES

The following pragma directives have been *added* in version 2.x:

```
#pragma constseg
#pragma dataseg
#pragma diag_default
#pragma diag_error
#pragma diag_remark
#pragma diag_suppress
#pragma diag_warning
#pragma inline
#pragma location
#pragma message
#pragma object_attribute
#pragma optimize
#pragma type_attribute
#pragma vector
```

Specific segment placement

In version 1.x, the `#pragma memory` directive supports a syntax that enables subsequent data objects that match certain criteria to end up in a specified segment. Each object found after the invocation of a segment placement directive will be placed in the segment, provided that it does not have a memory attribute placement, and that it has the correct constant attribute. For `constseg`, it must be a constant, while for `dataseg`, it cannot be declared `const`.

In version 2.x, the directive `#pragma location` and the `@` operator are available for this purpose.

Predefined symbols

All predefined symbols supported in version 1.x are supported also in version 2.x. version 2.x, however, have additional ones.

See the *M16C/R8C IAR C/C++ Compiler Reference Guide* for information about the predefined symbols.

Intrinsic functions

Version 1.x and version 2.x have different sets of intrinsic functions. In version 2.x, some intrinsic functions have been removed and some have been renamed.

REMOVED INTRINSIC FUNCTIONS

The following version 1.x intrinsic functions are *not available* in version 2.x:

```
__args$
__argt$
interrupt_on_overflow
overflow_flag_value
```

Note: Even though there is no `interrupt_on_overflow` function available in version 2.x, you can use `__RMPA_B_INTO` or `__RMPA_W_INTO` to combine an `RMPA` instruction with an `INTO` instruction. Likewise, even though the `overflow_flag_value` is not available, you can use `__overflow` after a call to `__RMPA_B_overflow` or `__RMPA_W_overflow` to find out if an `RMPA` instruction resulted in an overflow.

RENAMED INTRINSIC FUNCTIONS

The following table shows the intrinsic functions that have been *renamed* in version 2.x:

Intrinsic functions in version 1.x	Intrinsic functions in version 2.x
`break_instruction`	`__break`
`disable_interrupt`	`__disable_interrupt`
`enable_interrupt`	`__enable_interrupt`
`nop_instruction`	`__no_operation`
`read_ipl`	`__get_interrupt_level`
`rmpa_instruction`	`__RMPA_W`
`set_interrupt_table`	`__set_INTB_register`
`short_rmpa_instruction`	`__RMPA_B`
`software_interrupt`	`__software_interrupt`
`und_instruction`	`__illegal_opcode`
`wait_for_interrupt`	`__wait_for_interrupt`
`write_ipl`	`__set_interrupt_level`

Table 6: Version 1.x and version 2.x intrinsic functions

See the *M16C/R8C IAR C/C++ Compiler Reference Guide* for information about the intrinsic functions.

Other changes

This section describes changes related to:

- Object file format
- Nested comments
- Preprocessor file
- Cross-reference information
- `Sizeof` in preprocessor directives.

OBJECT FILE FORMAT

In version 1.x, two types of source references can be generated in the object file. When the command line option `-r` is used, the source statements are being referred to. When the command line option `-re` is used, the actual source code is embedded in the object format.

In version 2.x, when the command line option `-r` or `--debug` is used, source file references are always generated. Embedding of the source code is not supported.

NESTED COMMENTS

In the old version, nested comments are allowed if the option `-C` is used. In version 2.x, nested comments are never allowed. For example, if a comment was used for removing a statement as in the following example, it would not have the desired effect.

```
/*
/* x is a counter */
int x = 0;
*/
```

The variable `x` will still be defined, there will be a warning where the inner comment begins, and there will be an error where the outer comment ends.

```
  /* x is a counter */
  ^
"c:\bar.c",2  Warning[Pe009]: nested comment is not allowed

  */
   ^
"c:\bar.c",4  Error[Pe040]: expected an identifier
```

The solution is to use `#if 0` to “hide” portions of the source code when compiling:

```
#if 0
/* x is a counter */
int x = 0;
#endif
```

Note: `#if` statements may be nested.

PREPROCESSOR FILE

In version 1.x, a preprocessor file can be generated as a side effect of compiling a source file.

In version 2.x a preprocessor file is either generated as a side effect, or as the whole purpose when parsing of the source code is not required. You may also choose to include or exclude comments and/or `#line` directives.

SIZEOF IN PREPROCESSOR DIRECTIVES

In version 1.x, `sizeof` could be used in `#if` directives, for example:

```
#if sizeof(int)==2
int i = 0;
#endif
```

In version 2.x, `sizeof` is not allowed in `#if` directives. The following error message will be produced:

```
  #if sizeof(int)==2
      ^
"c:\bar.c",1  Error[Pe059]: function call is not allowed in a
constant expression.
```

Macros can be used instead, for example `SIZEOF_INT`. Macros can be defined using the `-D` option, or a `#define` in the source code:

```
#if SIZEOF_INT==2
int i = 0;
#endif
```

To find the size of a predefined data type, see the *M16C/R8C IAR C/C++ Compiler Reference Guide*.

Complex data types may be computed using one of several methods:

1 Write a small program and run it in the simulator, with terminal I/O.

```
#include <stdio.h>
struct s { char c; int a; };

void main(void)
{
  printf("sizeof(struct s)=%d \n", sizeof(struct s));
}
```

2 Write a small program, compile it with the option `-la .` to get an assembler listing in the current directory, and look for the definition of the constant `x`.

```
struct s { char c; int a; };
const int x = sizeof(struct s);
```