IAR Embedded Workbench®

Migration Guide

for Texas Instruments'
MSP430 Microcontroller Family

M430-2

IAR SYSTEMS

EDITION NOTICE

Second edition: March 2010

Part number: M430-2

This guide applies to version 5.x of IAR Embedded Workbench® for Texas Instruments' MSP430 microcontroller family.

Internal reference: M3, 6.0.x, IJOA.

Contents

Migrating to version 5.x from 3.x or 4.x

This chapter gives hints for porting your application code and projects to the new version 5.x from one of the old versions 3.x or 4.x.

Note that if you are migrating from an older version than 3.x or 4.x, you must first read the previous migration chapters in this guide.

Migration considerations

To migrate your old project, consider these topics:

- IAR Embedded Workbench IDE
- C language changes
- Runtime library changes.

Note that not all items in the list might be relevant for your project. Consider carefully which actions that are needed in your case.

Code written for version 3.x or 4.x might generate warnings or errors in version 5.x.

IAR Embedded Workbench IDE

When you upgrade to the new version of the IAR Embedded Workbench IDE, you must consider the issues described in this section.

INSTALLATION DIRECTORY

When you install your new version of the IAR Embedded Workbench IDE, you cannot install it in the same installation directory as your old version.

The old default installation path is:

`c:\Program Files\IAR Systems\Embedded Workbench 5.`*`n`*`\`

The new default installation path is:

`c:\Program Files\IAR Systems\Embedded Workbench 6.`*`n`*`\`

Note the difference in version number of the IDE. (This is not the same as the version number of your IAR Embedded Workbench product.)

PROJECT SETTINGS IN THE OPTIONS DIALOG BOX

The **Options** dialog box—available from the **Project** menu—has changed. This table lists the most important changes:

Category>Page	Changes
C/C++ Compiler>Language	See *C language changes*, page 6.
Linker	The tabs are listed in a new order.
Linker>Processing	The name of this page is now Checksum.

Table 1: Overview of changes in the Project options dialog box

PROJECT FILES

Even though some of the pages in the **Options** dialog box have changed, your old project file can be used in your new version of the IAR Embedded Workbench IDE.

C language changes

In version 5.x, the compiler by default conforms to the C99 standard (ISO/IEC 9899:1999 including technical corrigendum No.3), hereafter referred to as *Standard C* in this guide. Optionally, you can make the compiler conform to the C89 standard instead (ISO 9899:1990 including all technical corrigenda and addenda). In C89 mode, you cannot use any C99 language features or any C99 library symbols.

In version 3.x and 4.x, the compiler by default conforms to the C89 standard. Optionally, you can use some C99 features.

To migrate to version 5.x, you must consider:

- Options for language support
- Options for language conformance
- Obsolete C89 features in your source code.

OPTIONS FOR LANGUAGE SUPPORT

This table lists the differences in the options for enabling language support:

Language features	In version 5.x	In version 3.x/4.x
C89*	`--c89`†	Supported by default.
C99* (Standard C)	Supported by default.	Some features available when `-e` is used.

Table 2: Enabling language features

*** C89 and C99, respectively, but with some minor exceptions. For more information, see the compiler documentation.**
† `--c89` disables C99 library symbols and C99 language features.

Note: C99 mode does not allow variable length arrays (VLA) by default; use the command line option `--vla` to enable such support. Embedded C++ has not changed.

OPTIONS FOR LANGUAGE CONFORMANCE

The options for C/C++ language conformance differ between the two versions; this table lists these differences:

In version 5.x Option in IDE vs on the command line	In version 3.x/4.x Option in IDE vs on the command line	Description
Standard with IAR extensions `-e`	**Allow IAR extensions** `-e`	Accepts IAR extensions and IAR relaxations to Standard C.
Standard Supported by default on the command line	**Relaxed ISO/ANSI** Supported by default on the command line	Accepts IAR relaxations to Standard C.
Strict `--strict`	**Strict ISO/ANSI** `--strict_ansi`	Strict adherence to the standard.

Table 3: Options for language conformance

OBSOLETE C89 FEATURES IN YOUR SOURCE CODE

There are some C89 features that are accepted by the compiler in version 3.x and 4.x, but which are not accepted by the compiler in version 5.x when you compile in C99 mode. Warnings or errors will be generated. To omit these diagnostic messages, you must either compile the source code in C89 mode or rewrite your source code.

These C89 features are not accepted by the compiler in version 5.x when compiling in C99 mode:

- Implicit `int` variables

```
static k; /* k was implicit int. */
```

- Implicit `int` parameters

```
myFunction(i,j)
{
  /* i and j were implicit int. */
}
```

- Implicit `int` returns

```
myFunction()
{
  /* Returned implicit int 0. */
}
```

- Implicit returns

```
myFunction()
{
  /* Returned 0. */
}
```

- Implicit returns

```
myFunction()
{
  return; /* Returned 0. */
}
```

Runtime library changes

In version 5.x, the compiler and assembler automatically search for system header files in a predestined directory (relative to the compiler/assembler executable file). In version 3.x and 4.x, you must specify the include file search paths explicitly.

In version 5.x, these compiler options are available for this:

`--clib`	Uses CLIB system header files.
`--dlib`	Uses DLIB system header files.
`--dlib_config`	Uses DLIB system header files. The option also lets you specify the runtime library configuration to use. In version 3.x and 4.x, the option lets you specify a runtime library configuration file, but in version 5.x the option also accepts the argument `none`.

`--no_system_include`	Disables the automatic search for system header files. You must specify the include file search path explicitly, just as in version 3.x and 4.x. This option is useful if you have well-established script files for building your application project and you do not want to apply to the new include file system immediately.
`--system_include_dir`	Specifies the include directory explicitly, where the compiler can find the system header files.

These corresponding assembler options are available:

`-g`	Disables the automatic search for system header files. You must specify the include file search path explicitly, just as in version 3.x and 4.x. This option is useful if you have well-established script files for building your application project and you do not want to apply to the new include file system immediately.

For detailed reference information about these options, see the *IAR C/C++ Compiler Reference Guide for MSP430* and the *MSP430 IAR Assembler Reference Guide*.

Migrating to version 3.x or 4.x from version 2.x

This chapter gives hints for porting your application code and projects to version 3.x or 4.x. Read this chapter first if you are migrating from version 2.x. However, if you are migrating from the MSP430 IAR C Compiler version 1.x, you must first read the chapter *Migrating to version 2.x from version 1.x*.

C or C++ source code that was originally written for the MSP430 IAR C Compiler version 2.x can be used also with MSP430 IAR C/C++ Compiler version 3.x or 4.x. However, some small modifications might be required.

This chapter presents the major differences between the MSP430 IAR Embedded Workbench version 2.x and the MSP430 IAR Embedded Workbench version 3.x or 4.x, and describes the migration considerations.

Migration considerations

To migrate your old project consider the following:

- IAR Embedded Workbench IDE
- Runtime library and object files considerations.

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

Note: It is important to be aware of the fact that code written for version 2.x might generate warnings or errors in version 3.x or 4.x.

IAR Embedded Workbench IDE

Upgrading to version 3.x or 4.x of the IAR Embedded Workbench IDE should be a smooth process as the improvements do not affect the compatibility between the versions.

WORKSPACE AND PROJECTS

The workpaces and projects you have created with 2.x are compatible with 3.x and 4.x, except for debugger settings.

The first time you open an old project, it will be converted and a backup copy of the old project is saved unchanged. When you have opened and converted the project, you must review the settings and make any needed modifications in addition to making new debugger settings.

C-SPY® LAYOUT FILES

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

Runtime library and object files considerations

In version 3.x and 4.x, two sets of runtime libraries are provided—CLIB and DLIB. CLIB corresponds to the runtime library provided with version 2.x, and it can be used in the same way as before.

To build code produced by version 3.x or 4.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 14. For more information about the two libraries, and the runtime environment they provide, see the *IAR C/C++ Compiler Reference Guide for MSP430*.

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 MSP430 IAR Embedded Workbench version 3.x and 4.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. Perhaps an application uses the `fprintf` function for terminal I/O (`stdout`), but 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`. In other words, 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 defines the library functionality.

When you build an application using the IAR Embedded Workbench IDE, 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 (`r43`) there is a corresponding library configuration file (`h`), which has the same name as the library. The files are located in the `430\lib` directory. The command lines for specifying the library configuration file and library object file could look like this:

```
icc430 -D_DLIB_CONFIG_FILE=...\430\lib\dlib\dl430dn.h
xlink dl430dn.r43
```

To take advantage of the new features, it is recommended that you read about the runtime environment in the *IAR C/C++ Compiler Reference Guide for MSP430*.

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.

The new linker option **Entry label** (`-s`) specifies 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. As before, any program modules containing a root segment part will also be loaded.

In version 3.x and 4.x, the default program entry label in `cstartup.s43` 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.s43`.

If you build your application in the IAR Embedded Workbench IDE, you might simply add a 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 you link a C/C++ application. If you link without the option, the resulting output executable file will be empty, because no modules were referred to.

SYSTEM INITIALIZATION—CSTARTUP

The content of the `cstartup.s43` file has been split up into two files: `cstartup.s43` and `cexit.s43`.

Now, the `cstartup.s43` only contains exception vectors and initial startup code to setup stacks and processor mode. Note that the `cstartup.s43` file is the only one of these three files that might require any modifications.

The `cexit.s43` file contains termination code, for example, execution of C++ destructors.

For applications that use a modified copy of `cstartup.s43`, you must adapt it to the new file structure.

MIGRATING FROM CLIB TO DLIB

There are some considerations to have in mind 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 IDE to use the DLIB library.

You should also see the chapter *The DLIB runtime environment* in the *IAR C/C++ Compiler Reference Guide for MSP430*.

Migrating to version 2.x from version 1.x

This chapter contains information about migrating from the MSP430 IAR Embedded Workbench version 1.x to version 2.x. You must follow the instructions in this chapter first if you are migrating from version 1.x, because all the modifications described apply also to migration to later versions.

After you have read this chapter, you must read the chapter *Migrating to version 3.x or 4.x from version 2.x*, which contains information about migrating projects from the MSP430 IAR Embedded Workbench version 2.x to version 3.x or 4.x.

Differences

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

- In version 2.x, support for Embedded C++ became available
- 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 might generate warnings or errors in version 2.x.

The migration process

In short, to migrate from version 1.x to version 2.x, follow these steps:

1. If you have an IAR Embedded Workbench project file, you must first create a workspace in your IAR Embedded Workbench IDE version 2.x. Add your version 1.x project to the newly created workspace. The project will be automatically converted to a version 2.x project file. The version 1.x project file will still be available.
2. Set the appropriate compiler options.
3. Replace version 1.x extended keywords in the source code with keywords available in version 2.x.

4 Replace version 1.x pragma directives with directives available in version 2.x. Note that the behavior differs between the two products; see *Pragma directives*, page 21 for detailed information.

5 Verify the usage of intrinsic functions in the source code. For more information, see *Intrinsic functions*, page 23.

6 When building the project, make sure to use a linker command file supplied with version 2.x. If you are using the Embedded Workbench IDE interface, the appropriate linker command file is selected automatically when you choose **Project>Options>General Options** and select a device from the **Device** drop-down menu on the **Target** page. You might also consider updating any customizations made to the version 1.x linker command file.

Note: 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).

If you use the option `--migration_preprocessor_extensions`, version 2.x will accept such non-standard expressions.

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.

CONVERTING OPTION SETTINGS

Since the available options differ between version 1.x and version 2.x, you should verify your option settings after you have converted a version 1.x 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 that you verify the options manually. Follow these steps:

1. Open the version 1.x project in the IAR Embedded Workbench version 1.x.
2. In the project window, select the **Target** level.
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 version 1.x project are still available or needed. Also check whether you need to use any of the new options.

Removed options

The following table shows the command line options that were removed:

Version 1.x option	Description
`-C`	Nested comments
`-F`	Form-feed in list file after each function
`-G`	Open 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`	Add #include file text
`-K`	`'//'` comments; in version 2.x, `'//'` comments are allowed unless the option `--strict_ansi` is used
`-p`*nn*	Lines/page
`-T`	Active lines only
`-t`	Tab spacing
`-U`*symb*	Undefined preprocessor symbol
`-ua`	Enables odd address check
`-X`	Explain C declarations
`-x[DFT2]`	Cross-reference
`-y`	Writable strings

Table 4: Version 1 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`*	Define symbols
`-e`	Language extensions
`-f` *`filename`*	Extend the command line
`-I`	Include paths (The syntax is more free in version 2.x)
`-o` *`filename`*	Set object filename
`-s[0–9]`	Optimize for speed
`-z[0–9]`	Optimize for size

Table 5: Compiler options identical in both compiler versions

Note: For the optimization flags (`-s` and `-z`), only levels 2, 3, 6, and 9 are available in version 2.x.

Renamed or modified options

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

Version 1.x option	Version 2.x option	Description
`-A` `-a` *`filename`*	`-la .` `-la` *`filename`*	Assembler output; See *Filenames*, page 19
`-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 changed
`-r[012][i][n][v]`	`-r` `--debug`	Generates debug information; the modifiers were removed
`-S`	`--silent`	Sets silent operation

Table 6: Renamed or modified options

Version 1.x option	Version 2.x option	Description
-ur4	--lock_r4	Causes the compiler to generate ROM-monitor-compatible code by not using register R4
-ur5	--lock_r5	Causes the compiler to generate ROM-monitor-compatible code by not using register R5
-upic	--pic	Position-independent code, except function pointers
-ufp	--pic	Position-independent function pointers
-w	--no_warnings	Disables warnings

Table 6: Renamed or modified options (Continued)

Note: A number of new command line options were added.

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:

Version 1.x command	Version 2.x command	Result
-l myfile	-l myfile	myfile.lst
-Lmyfile	-l myfiletest	myfiletest.lst
-L	-l .	test.lst
-Lmydir/	-l mydir	mydir/test.lst

Table 7: 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. In version 2.x, the 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 extended keywords changed in version 2.x. Some keywords were added, some keywords were removed, and for some of them the syntax changed. In addition, one 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 `__no_init`.

The following table lists the version 1.x keywords, their version 2.x equivalents, and additional keywords:

Version 1.x keyword	Version 2.x keyword
–	`asm`
`interrupt`	`__interrupt`
`monitor`	`__monitor`
`no_init`	`__no_init`
`sfrb`	–
`sfrw`	–

Table 8: Version 1.x and version 2.x extended keywords

INTERRUPT FUNCTIONS AND VECTORS

The syntax for defining interrupt functions changed from version 1.x.

Version 1.x syntax

The syntax when defining interrupt functions using version 1.x:

```
interrupt [vector] void function_name(void);
```

where *vector* is the vector offset in the vector table.

Version 2.x syntax

The syntax when defining interrupt functions using version 2.x:

```
#pragma vector=vector
__interrupt void function_name(void);
```

where *vector* is the vector offset in the vector table.

ABSOLUTE LOCATED VARIABLES

In version 2.x you can:

- 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:

```
__no_init volatile char PORT @ 100;
```

The extended keywords `sfrb` and `sfrw` are not available in version 2.x.

Version 1.x syntax

```
sfrb P0IN = 0x10;
```

Version 2.x syntax

```
__no_init volatile unsigned char P0IN @ 0x10;
```

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` specified the default location of data objects, and `#pragma function` specified the default location of functions. They changed the default attribute to use for declared objects; they did 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`.

The following pragma directives were removed from version 1.x:

- `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, the `#pragma location` directive or the `@` operator can be used for specifying an absolute location.

The following table shows the mapping of pragma directives:

Version 1.x directive	Version 2.x directive
`#pragma function=interrupt`	`#pragma type_attribute=__interrupt`
`#pragma function=monitor`	`#pragma type_attribute=__monitor`
`#pragma memory=constseg`	`#pragma constseg, #pragma location`
`#pragma memory=dataseg`	`#pragma dataseg, #pragma location`
`#message`	`#pragma message`

Table 9: Version 1.x and version 2.x pragma directives

It is important to note that the version 2.x 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;
```

Specific segment placement

In version 1.x, the `#pragma memory` directive supported a syntax that enabled subsequent data objects that matched certain criteria to end up in a specified segment. Each object found after the invocation of a segment placement directive was placed in the segment, provided that it did not have a memory attribute placement, and that it had the correct constant attribute. For `constseg`, it had to be a constant, while for `dataseg`, it could not be declared `const`.

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

Diagnostics

Instead of using the removed version 1.x pragma directive `warnings`, you can specify diagnostics options on the compiler command line (see the *Compiler options* chapter in the *IAR C/C++ Compiler Reference Guide for MSP430*) or by choosing **Project>Options>C/C++ Compiler>Diagnostics** in the IDE.

Intrinsic functions

Version 2.x has a new naming convention for intrinsic functions, as well as a large number of additional functions.

The intrinsic functions `_args$` and `_argt$` available in version 1.x were removed and cannot be used in version 2.x. However, except for these two functions, all intrinsic functions available in version 1.x can be used also in version 2.x.

To use the version 1.x intrinsic functions, include the file `in430.h`. To use the version 2.x intrinsic functions, include the file `intrinsic.h`.

The following table lists the version 1.x intrinsic functions and their version 2.x equivalents, as well as the new intrinsic functions in version 2.x:

Version 1.x intrinsic function	Version 2.x intrinsic function	Description
`_args$`	–	Returns an array of the parameters to a function.
`_argt$`	–	Returns the type of a parameter.
`_BIC_SR`	`__bic_SR_register`	Clears a bit in the status register.
`_BIC_SR_IRQ`	`__bic_SR_register_on_exit`	Ensures that bits are cleared in the processor status register when an interrupt or monitor function returns.
`_BIS_NMI_IE1`	–	Ensures that bits are set in the interrupt enable control bits 1 (`IE1 address 0x0000`) when the current interrupt or monitor function returns.
`_BIS_SR`	`__bis_SR_register`	Sets a bit in the status register.
`_BIS_SR_IRQ`	`__bis_SR_register_on_exit`	Ensures that bits are set in the processor status register when an interrupt or monitor function returns.
`_DINT`	`__disable_interrupt`	Disables interrupts by inserting the `DINT` instruction.
`_EINT`	`__enable_interrupt`	Enables interrupts by inserting the `EINT` instruction.
–	`__get_interrupt_state`	Returns the current interrupt state.

Table 10: Version 1.x and version 2.x intrinsic functions

Version 1.x intrinsic function	Version 2.x intrinsic function	Description
–	`__get_R4_register`	Returns the value of the R4 processor register.
–	`__get_R5_register`	Returns the value of the R5 processor register.
-	`__get_SR_register`	Returns the value of the processor status register.
-	`__get_SR_register_on_exit`	Returns the value of the of the processor status register when the current interrupt or monitor function returns.
–	`__low_power_mode`*n*	Enters the MSP430 low power modes, where *n* can be one of 0–4.
–	`__low_power_mode_off_on_exit`	Turns off low power mode when monitor or interrupt function returns.
`_NOP`	`__no_operation`	Generates a `NOP` instruction.
`_OPC`	`__op_code`	Inserts a `DW const` directive.
–	`__segment_begin`	Returns the start address of a segment.
–	`__segment_end`	Returns the end address of a segment.
–	`__set_interrupt_state`	Sets the current interrupt state.
–	`__set_R4_register`	Sets the value of the R4 processor register.
–	`__set_R5_register`	Sets the value of the R5 processor register.
`_SWAP_BYTES`	`__swap_bytes`	Executes the `SWPB` instruction.

Table 10: Version 1.x and version 2.x intrinsic functions (Continued)

Segments

The segment naming convention changed since version 1.x. Some of the version 1.x segments disappeared, and some new were introduced.

This table lists the version 1.x segment names, their counterparts in version 2.x, and additional segments:

Version 1.x segment	Version 2.x segment
CCSTR**	–
CDATA0	DATA16_ID
CODE	CODE
CONST	DATA16_C
CSTACK	CSTACK
CSTR	DATA16_C
ECSTR**	–
IDATA0	DATA16_I
INTVEC	INTVEC
NO_INIT	DATA16_N
UDATA0	DATA16_Z
–	DATA16_AC*
–	DATA16_AN*
–	HEAP
–	DIFUNC

Table 11: Version 1.x and version 2.x segments

Note: * Segments ending in _AN and _AC contain data located at absolute addresses, and should not be included in the linker command file.

Note: ** Version 2.x does not support placing strings in writable memory. Hence, the version 1.x segments used for this task have no counterparts in version 2.x.

Assembler source code

If you have used any of the segments specific to version 1.x in assembler source code, and if you want to port this assembler source code, you must replace all version 1.x segment names with version 2.x segment names.

If your application is written entirely in assembler, you must use the option **Ignore CSTARTUP in library** which can be found on the **Include** options page in the **XLINK** category in version 2.x of the MSP430 IAR Embedded Workbench IDE.

Other changes

This section describes changes related to:

- Object file format
- Predefined symbols
- Nested comments
- `Sizeof` in preprocessor directives
- Include files
- Runtime library.

OBJECT FILE FORMAT

In version 1.x, two types of source references could be generated in the object file. When the command line option `-r` was used, the source statements were being referred to, and when the command line option `-re` was used, the actual source code was 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, that is, embedding of the source code is not supported.

PREDEFINED SYMBOLS

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

The predefined symbol `__IAR_SYSTEMS_ICC` is provided only for compatibility with version 1.x. Version 2.x also has the `__IAR_SYSTEMS_ICC__` symbol.

NESTED COMMENTS

In version 1.x, nested comments were allowed if the option `-C` was used. In version 2.x, nested comments are never allowed. For example, if a comment is used for removing a statement as in the following example, it will 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 can be nested.

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 *IAR C/C++ Compiler Reference Guide for MSP430*.

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

- 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));
}
```

- 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);
```

INCLUDE FILES

Version 2.x has two types of include files for SFR definitions:

`io430xxx.h`	New include files for I/O definitions with bitfields for bit access of SFRs
`msp430xxx.h`	Provided for compatibility with version 1.x

RUNTIME LIBRARY

To build code produced by version 2.x, you must also use the runtime libraries it provides. That is, it is not possible to link object code produced using version 2.x with a runtime library provided with version 1.x.

In version 2.x, two sets of runtime libraries are provided—CLIB and DLIB. CLIB corresponds to the runtime library provided with version 1.x. For more information about the two libraries and the runtime environment, see the *IAR C/C++ Compiler Reference Guide for MSP430*.

Support for the hardware multiplier

In version 1.x a dedicated runtime-library object file—`cl430m.r43`—had to be used to support the hardware multiplier peripheral unit.

To enable support for the hardware multiplier in later versions, see the *IAR C/C++ Compiler Reference Guide for MSP430* or the *IAR Embedded Workbench® IDE for MSP430 User Guide*.

Floating-point arithmetic

In version 1.x, there was only support for 32-bit floating-point arithmetic. In version 2.x, there is support for both 32- and 64-bit floating points.

In version 2.x, the IAR CLIB library had two different floating-point implementations—one fully IEEE754-compliant and one not fully IEEE754-compliant but instead more compact.