IAR Embedded Workbench

Migration Guide

for Arm Limited's

Arm® Cores

MARM-6

IAR SYSTEMS

EDITION NOTICE
Sixth edition: November 2021

Part number: MARM-6

This guide applies to version 9.20.x of IAR Embedded Workbench® for Arm.

Internal reference: BB9, FF9.0.8, IJOA.

Migrating from version 8.x to 9.x

This chapter gives hints for porting your application code and projects to the new version 9.x from the old version 8.x of IAR Embedded Workbench for Arm.

Note that if you are migrating from a version older than 8.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-SPY changes
- Runtime library changes
- MISRA C compiler changes

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

SYSTEM REQUIREMENTS

The IAR Embedded Workbench for Arm version 9.x requires a 64-bit Windows 7 or Windows 10 operating system. These changes have been made:

- All incorporated Embedded Workbench Windows executable files now run as Windows 64-bit applications
- The 32-bit Windows operating system is no longer supported.

COMPILER ERRORS FOR NON-STANDARD C/C++

In version 9.x, the compiler can issue new errors or warnings for illegal constructs that are not legal according to the C/C++ standard.

Note: The compiler will generate a warning for the illegal C/C++ construct where you first declare a symbol as having an external linkage and then declare it as static.

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 (x86)\IAR Systems\Embedded Workbench 8.`*`n`*`\`

The new default installation path is:

`c:\Program Files\IAR Systems\Embedded Workbench 9.`*`n`*`\`

Note the difference in the version number of IAR Embedded Workbench.

NO BUILD STATUS IN WORKSPACE WINDOW

In version 9.x, the **Workspace** window no longer includes a column that shows whether a file will be rebuilt next time the project is built.

PATHS TO EXECUTABLE FILES IN THE IDE

In version 9.x, paths to executable files that contain whitespace must be surrounded by quote marks to avoid ambiguities.

PROJECTS BUILT WITH PREBUILT CMSIS DSP LIBRARIES

CMSIS version 5.7.0 is the last version that was delivered with prebuilt DSP libraries, to be included by choosing **Project>Options>General Options>Library Configuration>CMSIS>DSP library**.

To build a project with DSP libraries and use features from CMSIS version 5.8.0 or later, or build for targets that were not available when 5.7.0 was released, you must create an empty CMSIS project and manually add your application code to it, instead of using the **DSP library** option on the **Library Configuration** option page.

Then, in the **CMSIS Manager Components** window, select **DSP** in the **CMSIS** category (available after you have installed the ARM.CMSIS pack).

Note: The 5.7.0 DSP libraries are still available for building legacy projects and targets.

C-SPY changes

C-SPY EXPRESSION #PC

In version 9.x, the expression `#PC` is ambiguous in C-SPY macros on 64-bit architectures. Therefore, when you debug a 64-bit MCU, you must use `#PC32` for AArch32 and `#PC64` for AArch64 in C-SPY macros. When debugging a 32-bit MCU, use `#PC`.

Runtime library changes

DEFAULT TIME HANDLING IN DLIB RUNTIME LIBRARY

In version 9.x, the default for time handling in the DLIB runtime library has changed from 32-bit to 64-bit. These changes have been made:

- `time.h` uses 64-bit time handling
- `time_t` is 64-bit
- `clock_t` is 8 bytes if `long` is 8 bytes and 64-bit `time.h` is used, otherwise it is 4 bytes
- `__time()` uses 64-bit `time_t`
- `__clock()` uses 64-bit `clock_t`

Note: It is the default behavior for time handling that has changed to 64-bit. You can still choose 32-bit time handling as described in the *IAR C/C++ Development Guide for Arm*.

Note: The linker does not produce a message when linking with old object modules that have 32-bit time handling versions of `__time()` or `__clock()`. The linker defaults to 64-bit handling.

For more information about `time.h`, see the *IAR C/C++ Development Guide for Arm*.

Using default time_t and __time32 function in 8.50.9 or earlier

There is an issue if you used the default `time_t` and implemented the `__time32` function in version 8.50.9 or earlier.

When you upgrade the project to version 9.x, the `__time32` function is no longer called, and the default `__time64` is used instead. There is no warning. Because `__time32` and `__time64` are low-level functions that return the current calendar time (customized by customers for each board), previous time/calendar functionality will fail without warning (`__time64` is called instead of `__time32`). The default `__time64` function returns `-1`.

MISRA C compiler changes

MISRA C:1998/C:2004 CHECKER REMOVED

In version 9.x, the compiler's MISRA C:1998/C:2004 checker has been removed. For MISRA C support, you must use C-STAT, which gives a much larger selection of checks.

Note: In version 9.x, the compiler's MISRA C:1998/C:2004 checker is still available through the compiler command line, but will be removed in a future release.

Migrating from version 6.x or 7.x to 8.x

This chapter gives hints for porting your application code and projects to the new version 8.x from the old versions 6.x and 7.x of IAR Embedded Workbench for ARM®.

Note that if you are migrating from a version older than 6.x, you must first read the previous migration chapters in this guide.

There are no migration issues when migrating from version 6.x to 7.x.

Migration considerations

To migrate your old project, consider these topics:

- IAR Embedded Workbench IDE
- C/C++ language changes
- Changed options
- Library structure changes

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

Note: Code written for version 6.x or 7.x might generate warnings or errors in version 8.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 6.`*`n`*`\`

or

`c:\Program Files\IAR Systems\Embedded Workbench 7.n\`

The new default installation path is:

`c:\Program Files (x86)\IAR Systems\Embedded Workbench 8.n\`

Note the difference in the version number of IAR Embedded Workbench.

C/C++ language changes

C

In version 8.x, the compiler by default conforms to the C11 standard (ISO/IEC 9899:2012), hereafter referred to as *Standard C* in this guide.

Any old object files that were compiled using older versions of the C language, that used any system headers in non-AEABI mode, must be recompiled.

C++

In version 8.x, the compiler uses the C++14 standard (ISO/IEC 14882:2014(E)), hereafter referred to as *Standard C++* in this guide, instead of the old C++03 standard.

Any old object files that were compiled using the C++03 standard must be recompiled.

Source files written in the C++03 standard will likely compile as intended using the C++14 language. There are some syntax changes made to the C++14 standard that can make a C++03 source file either not compile or even fail at runtime. See the Standard C++ document, *appendix C.2*.

Examples:

- `X<1>>2>>x;` is now illegal
- `auto` now means type deduction and not stack placement.

EMBEDDED C++

Embedded C++ (EC++) and Extended Embedded C++ (EEC++) are no longer supported. The update of an IAR Embedded Workbench project will automatically change the **EC++** or **EEC++** language option to **C++**. On the command line, you must change the option `--ec++` and `--eec++` into `--c++ --no_exceptions --no_rtti`.

Any old object files that were compiled using the EC++ or EEC++ language must be recompiled.

Source files written in the EC++ or EEC++ language must be ported to Standard C++. The library symbols now reside in the `namespace std`. A remedy for this is to either prefix references to library symbols with `std::` or to insert `using namespace std;` after the last inclusion of a C++ system header. See also *C++*, page 8.

Changed options

MULTIBYTE SUPPORT

The compiler option `--enable_multibytes` has been removed. The option determined the following:

- whether the compiler should use the Raw encoding or the system default locale encoding for string literals, character constants, and comments in the source file.
- whether the automatic choice of a formatter variant for `printf` and `scanf` should support multibytes and `wchar_t` formatters.

The equivalent options to consider are:

- None, if the Raw encoding should be used for the source file.
- `--source_encoding locale`, if the system default locale should be used for the source file.

If you update your IAR Embedded Workbench project, setting the option `--source_encoding` will be handled automatically.

Use the compiler options `--printf_multibytes` and `--scanf_multibytes` to control whether the automatic choice of `printf` and `scanf` formatter should support multibytes or not.

PREPROCESSOR OUTPUT

The compiler option `--preprocess` now has a different set of parameters:

`c`	Include comments
`n`	Preprocess only
`s`	Suppress `#line` directives

Project>Options>C/C++ Compiler>Preprocessor>Preprocessor output to file

LANGUAGE OPTIONS

--ec++

The compiler option `--ec++` has been removed. Use `--c++` with `--no_exceptions` and `--no_rtti` instead.

Project>Options>C/C++ Compiler>Language 1>C++

--eec++

The compiler option `--eec++` has been removed. Use `--c++` with `--no_exceptions` and `--no_rtti` instead.

Project>Options>C/C++ Compiler>Language 1>C++

Library structure changes

These changes have been made to the structure of the library source code:

- The system header `yfuns.h` is now named `LowLevelIOInterface.h`.
- The defines `_STD_BEGIN` and `_STD_END` have been removed. Remove them from your source code or define them to nothing.

Migrating from version 5.x to 6.x

This chapter gives hints for porting your application code and projects to the new version 6.x from the old version 5.x of IAR Embedded Workbench for ARM®.

Note that if you are migrating from a version older than 5.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/C++ language changes
- Runtime library changes

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

Note: Code written for version 5.x might generate warnings or errors in version 6.x.

CHANGES IN USER DOCUMENTATION

The *IAR Embedded Workbench® IDE User Guide for ARM®* has been replaced by:

- The *C-SPY® Debugging Guide for ARM®*, which replaces all information related to debugging
- The *IDE Project Management and Building for ARM®*, which replaces all information related to project management and building in the IDE.

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 the version number of IAR Embedded Workbench.

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/C++ language changes*, page 12.

Table 1: Overview of changes in the 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.

When you convert an old project, the new option **C++ inline semantics** will be enabled. If you want your source code to be compliant to Standard C (C99), make sure to disable this option. For more information, read about **--use_c++_inlines** in the *IAR C/C++ Development Guide for ARM®*.

C/C++ language changes

In version 6.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 5.x, the compiler by default conforms to the C89 standard. Optionally, you can use some C99 features.

To migrate to version 6.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 6.x	In version 5.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` or **Allow VLA** in the IDE to enable such support.

OPTIONS FOR LANGUAGE CONFORMANCE

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

In version 6.x Option in IDE vs on the command line	In version 5.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

EMBEDDED C++ AND STANDARD C++

Standard C++ is supported in version 6.x. For more information about using Standard C++, see the *IAR C/C++ Development Guide for ARM®*. There is no automatic way to convert an old Embedded C++ project to a Standard C++ project. Note also that the EC++ library is not compatible with the C++ library. The implementation of Embedded C++ has not changed.

C99 INLINE

In version 5.x, an inline version of a function should have the same code wherever it is defined. At link time, any of them can be used as the non-inlined version. In C99, an inline function can have different code in every location where it is defined, but at link time there can only be one non-inlined version (the one declared `extern inline`). For example:

```
static int x;
inline void f()
{
  //static int y; // Can not refer to statics.
  //x;            // Can not refer to statics.
}

// Declare this f as the non-inlined version to use.
extern inline void f();
```

OBSOLETE C89 FEATURES IN YOUR SOURCE CODE

There are some C89 features that are accepted by the compiler in version 5.x, but which are not accepted by the compiler in version 6.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 6.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 6.x, the compiler and assembler automatically search for system header files in a predestined directory (relative to the compiler/assembler executable file). In version 5.x, you must specify the include file search paths explicitly.

In version 6.x, these compiler options are available for this:

`--dlib_config` *token*	Uses DLIB system header files. The option also lets you specify the runtime library configuration to use. In version 5.x, the option lets you specify a runtime library configuration file, but in version 6.x the option also accepts tokens.
`--no_system_include`	Disables the automatic search for system header files. You must specify the include file search path explicitly, just as in version 5.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 5.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 `inc` directory explicitly, where the assembler can find the system header files.

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

Migrating from version 4.x to version 5.x

This guide presents the major differences between ARM IAR Embedded Workbench® version 4.x and ARM IAR Embedded Workbench version 5.x, and describes the migration considerations. Primarily, these include how to:

- Make your existing application source code compile and link successfully
- Identify potential changes in runtime behavior.

This guide helps you to port your application source code and other project files to the new version 5.x.

Note that if you are migrating from ARM IAR Embedded Workbench version 3.x, you must first read the chapter *Migrating to Arm® IAR Embedded Workbench version 4.x*.

The migration process

The main conceptual difference between version 4.x and version 5.x is that the internal object format used by the IAR build tools has changed. In version 4.x, the IAR Systems format UBROF is used, whereas version 5.x uses the industry-standard format *Executable and Linking Format* including DWARF for debug information (ELF/DWARF). This implies a completely new linker—the IAR ILINK Linker—which replaces the IAR XLINK Linker.

The object format has changed to be compatible with tools from other vendors that also support ELF/DWARF.

The differences force you to modify your application source code and other related project files. In short, to migrate from version 4.x to 5.x, you must consider changes in the:

- Compiler and C source code
- Assembler and assembler source code
- Linker and linker configuration
- Runtime environment and object files

- Project files and project setup in the IAR Embedded Workbench IDE
- Debugger.

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. Consider carefully what actions are needed in your case.

Note: In version 5.x:

- The IAR Systems applications XAR and XLIB for maintaining libraries have been replaced by the IAR Archive Builder—`iarchive`—and a set of GNU utilities, also provided with the IAR product installation.
- The structure of the user documentation has changed. Earlier, the compiler and linker were documented in two separate guides. Now, the compiler and linker are both documented in the *IAR C/C++ Development Guide for ARM®*.

Compiler and C source code

C or C++ source code that was originally written for the ARM IAR C/C++ Compiler version 4.x can also be used with the new ARM IAR C/C++ Compiler version 5.x. However, before using the new compiler to compile existing source code, you should check the following details:

In your C/C++ source code files, be aware of the following changes:

- Initializers are no longer allowed for absolute placed constants, which means the following type of constructions are no longer allowed:

  ```
  int const a @ 10 = 20;
  ```

- The `#pragma vector` directive is no longer available. Interrupt vectors now have predefined names, for example `IRQ_Handler` as defined in `cstartup.o`.
- The `#pragma swi_number` directive and the `__swi` keyword are now only available for function declaration, not for function definition.
- The intrinsic functions `__enable_interrupt` and `__disable_interrupt` are no longer available as intrinsic functions; however, they are available as library functions, which means they are backwards compatible on source code level.

- The `__monitor` keyword is no longer available. You can replace the keyword with the following code sequence:

```
void function_that_was_declared_monitor_in_ewarm_4xx()
{
  __istate_t isate = __get_interrupt_state();
  __disable_interrupt();
  ...
  __set_interrupt_state(istate);
}
```

- The `#pragma optimize` directive has changed behavior. The following parameters are recognized but ignored in version 5.x: `s` for speed, `z` for size, `2` for none, `3` for low, `6` for medium, and `9` for high. In version 5.x, they have been replaced by the parameters `speed`, `size`, and `balanced`, where the latter balances between speed and size. They can be combined with the parameter `high`.
- The segment operator `__segment_size` is no longer available. To replace this operator, you can use the following type of construction:

```
size = __section_end("xxx") - __section_begin("xxx");
```

- The default layout for bitfields has changed from `disjoint_types` to `joint_types`. However, for migration you only have to pay attention to bitfields that are part of an external interface. In this case, refer to the *IAR C/C++ Development Guide for ARM®* for information about bitfields.

  Note that in little-endian mode, the layout for structures where all bitfields have the same base type is the same as before.

2 Instead of segments, the compiler now places code and data in sections. This internal change does not require any changes in your C/C++ source code, unless you are using any predefined segment names explicitly in your source code. In that case, you must make sure to use the new section names, see *Segments versus sections*, page 31.

Also, the handling of initialized segments has changed, see *Segments for initialization*, page 31.

3 There are some changes related to the compiler options. Some options have been removed, some options have changed, and there are some new options. For a list of changes, see *Tools options*, page 27.

4 For information about changes related to filename extensions, see *Filename extensions*, page 34.

5 The directory structure for I/O definition header files has changed. In version 5.x, these files are placed in device-specific subfolders. This affects the search path, which means you must modify the header file accordingly. For example:

```
#incude <iom1671000.h>
```

should be changed to:

```
#incude <oki/ioml671000.h>
```

For more details about the functionality in version 5.x, refer to the *IAR C/C++ Development Guide for ARM®*.

Assembler and assembler source code

The name of the assembler executable file has been renamed from `aarm` to `iasmarm`.

In your assembler source code, consider the following issues:

1. Modules

   In version 5.x, neither the assembler nor the compiler can make a distinction between program and library *modules*. If you want a module to be treated as a library module, thus conditionally linked, you must place the module in a library.

   This means that if you have used either the `LIBRARY` or the `MODULE` directive in your existing assembler source code, these will no longer have any effect.

   In version 4.x, you could define one or several assembler modules in each file. In version 5.x, there can only be one module per file. This means that you have to restructure your files accordingly.

   To read more about modular programming and the new syntax of the module directives. see the *ARM® IAR Assembler Reference Guide.*

2. Segments versus sections

   The segment concept has been replaced by the concept of sections. This means that:

   - Assembler directives operating on segments have been either removed or replaced by new directives operating on sections instead, which means you must modify your assembler source code accordingly. For more information, read about section control directives in the *ARM® IAR Assembler Reference Guide.*
   - If you have used any of the predefined segments specific to version 4.x in assembler source code, you must replace all old segment names with new section names. For further details, see the section *Segments versus sections*, page 31.

3. Expressions

   In version 5.x, it is not possible to have two symbols in one expression, or any other complex expressions, unless the expression can be resolved at assembly time. Any such

expressions must be rewritten, otherwise the assembler will generate an error. The following examples list expressions that cannot be solved at link time:

```
        public glob_var
        extern ext_var
var1    DC32    glob_var + ext_var   ; will fail
var2    DC32    glob_var * 5 + 3     ; will fail
var3    DC32    glob_var + 3         ; OK
```

4 Assembler directives

Some of the assembler directives have been removed and some use a new syntax or have other changes. For a list of assembler directives which are not the same in 5.x as in version 4.x, see *Assembler directives*, page 32.

If you have used any of these directives in your assembler source code, you must rewrite these constructions.

For detailed information about these directives, see the *ARM® IAR Assembler Reference Guide*.

5 Predefined symbols

The predefined symbol `__ASMARM__` has been replaced by the symbol `__IASMArm__`.

6 Calling convention

The calling convention used by the compiler has changed. In version 4.x, the stack is by default aligned to 4, but it is possible to align the stack to 8 according to the Advanced RISC Machines Ltd Arm/Thumb Procedure Call Standard (ATPCS). In version 5.x, the compiler instead follows the ARM Architecture Procedure Call Standard (AAPCS). This means that the stack is now aligned to 8 bytes.

This means that if you have C functions calling assembler functions, or vice versa, you must rewrite your assembler routines so that they follow the new calling convention.

To read more about these procedure call standards and the differences between them, refer to the *IAR C/C++ Development Guide for ARM®*, but also to the `www.arm.com` web site.

7 Backtrace information for the C-SPY Call stack window

The compiler resource names for backtrace information in the C-SPY Call Stack window have been standardized, and are defined in `CfiCommon.h`. This means that you can no longer define your own resource names. If you have used the `CFI` assembler directive to define your *names object*, this must contain a subset of the standardized resource names. For a list of the standardized resource names, see the *IAR C/C++ Development Guide for ARM®*.

8 For information about changes related to filename extensions, see *Filename extensions*, page 34.

9 The environment variables `ASMARM` and `Aarm_INC` have changed to `IASMarm` and `IASMarm_INC`, respectively.

Linker and linker configuration

The IAR XLINK Linker has been replaced by the IAR ILINK Linker.

XLINK VERSUS ILINK

Both XLINK and ILINK combine one or more relocatable object files with selected parts of one or more object libraries to produce an executable image. XLINK can only take object files in UBROF format, produced by tools from IAR Systems and produce output in the output format UBROF or in any of the other supported output formats. ILINK can take object files in ELF format and produces an executable image in the ELF format.

In version 4.x, the compiler places code and data in UBROF *segments*, which XLINK allocates in memory according to directives specified in the *linker command file*. This file is an extension of the command line, which means that you can simply specify any XLINK command line option in it. In version 5.x, the compiler places code and data in ELF *sections*. ILINK allocates these sections according to the *configuration* specified in the *ILINK configuration file*. This file also supports automatic handling of the application's initialization phase, which means initializing global variable areas and code areas by copying initializers and possibly decompressing them as well. However, the file cannot contain any command line options; these must be specified on the command line.

MIGRATING FROM XLINK TO ILINK

1 The new IAR ILINK Linker is target-specific and it replaces the IAR XLINK Linker; the name of the executable file has been renamed from `xlink` to `ilinkarm`.

2 To migrate your linker command file to a new ILINK configuration file, see *Converting XLINK.xcl to ILINK.icf*, page 23 for an example.

3 For information about how to map segments to sections, see *Segments versus sections*, page 31.

4 For information about changes related to filename extensions, see *Filename extensions*, page 34.

5 Use the provided GNU utilities if you need to convert the ELF output to either Intel-hex or Motorola S-records.

To learn about the more advanced features of ILINK, read the chapters about linking available in the *IAR C/C++ Development Guide for ARM®*.

CONVERTING XLINK.XCL TO ILINK.ICF

Because the linker command file (XLINK) and the linker configuration file (ILINK) are based on two different paradigms, nothing in the linker command file is automatically converted. Instead, you have to convert your linker setup manually.

If you are using the IAR Embedded Workbench IDE, you can use the linker configuration file editor to set up your linker configuration.

1. Choose **Project>Options**, select the **Linker** category and then click the **Config** tab.
2. To open the linker configuration file editor, select the **Override default** option and click the **Edit** button.
3. In the dialog box that appears you can define the:
   - Start address for the interrupt vector table
   - Start and end addresses for the RAM and ROM memory regions
   - Sizes of the stacks and heaps.
4. When you are finished, click the **Save** button. When you do this for the first time, a **Save As** dialog box appears.

   Note: You must explicitly select a dedicated linker configuration file for all your build configurations.

If you build your project from the command line, you can use the linker configuration file `generic.icf` located in the `arm\config` directory or any of the configuration files that are available in the example projects located in the `examples` directory. You can use any of these configuration files as a template for creating a configuration file that suits your target hardware and application requirements.

As an example, below is an XLINK linker command file (`xcl`) and a corresponding ILINK linker configuration file (`icf`).

```
-!========================
-!xlink  xcl file
-!========================
-carm

-DROMSTART=08000
-DROMEND=FFFFF
```

```
-Z(CODE)INTVEC=00-3F
-Z(CODE)ICODE,DIFUNCT=ROMSTART-ROMEND
-Z(CODE)SWITAB=ROMSTART-ROMEND
-Z(CODE)CODE=ROMSTART-ROMEND
-Z(CONST)CODE_ID=ROMSTART-ROMEND
-Z(CONST)INITTAB,DATA_ID,DATA_C=ROMSTART-ROMEND
-Z(CONST)CHECKSUM=ROMSTART-ROMEND

-DRAMSTART=100000
-DRAMEND=7FFFFF

-Z(DATA)DATA_I,DATA_Z,DATA_N=RAMSTART-RAMEND
-Z(DATA)CODE_I=RAMSTART-RAMEND
-QCODE_I=CODE_ID

-D_CSTACK_SIZE=2000
-D_IRQ_STACK_SIZE=100
-D_HEAP_SIZE=8000

-Z(DATA)CSTACK+_CSTACK_SIZE=RAMSTART-RAMEND
-Z(DATA)IRQ_STACK+_IRQ_STACK_SIZE,HEAP+_HEAP_SIZE=RAMSTART-RAMEND
```

This is the corresponding `icf` file:

```
//=========================
//ilink icf file
//-carm is not relevant to migrate because ilinkarm is
//ARM-specific.
//=========================
define memory mem with size = 4G;

define region ROM_region = mem:[from   0x8000 to  0xFFFFF];
define region RAM_region = mem:[from 0x100000 to 0x7FFFFF];

initialize by copy { rw };
do not initialize  { section .noinit };

define block CSTACK    with alignment = 8, size = 0x2000 { };
define block IRQ_STACK with alignment = 8, size =  0x100 { };
define block HEAP      with alignment = 8, size = 0x8000 { };

place at address mem:0x0 { ro section .intvec };
place in ROM_region      { ro };
place in RAM_region     { rw, block CSTACK, block IRQ_STACK,
                          block HEAP };
```

Project files and project setup in the IAR Embedded Workbench IDE

Upgrading to the new version of the IAR Embedded Workbench IDE requires some manual adaptations.

CONVERTING YOUR PROJECT FILE

If you are using the IAR Embedded Workbench IDE, start your new version of the ARM IAR Embedded Workbench IDE and open your old workspace. When you open a workspace that contains old projects created with version 4.x, a dialog box asks you if you want the project file to be converted for version 5.x. If you click **OK**, a backup of your old project is first created, and then the project is converted.

MIGRATING PROJECT OPTIONS

Because the available tools options differ between version 4.x and version 5.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 mapping tables in *Tools options*, page 27, and modify the makefile accordingly.

If you are using the IAR Embedded Workbench IDE, the options that are the same in both versions are automatically converted during the project conversion. The options that have changed will be set to default values.

To verify the options manually, follow these instructions:

- **Compiler** category

  The **Code** page is new, but the options were earlier available in the **General** category. The options will keep their settings.

  The **Optimizations** page has changed, because the `-O` compiler option replaces the `-s` and `-z` options, see *-s and -z versus -O*, page 28.

  The **Output** page has changed. If you have defined your own segment name, this will not be automatically converted to a section name. The default code section name is `.text`. For more information about segment versus section names, see *Segments versus sections*, page 31.

- **Linker** category

  No linker options are converted automatically. During the project conversion, all linker options will be set to default values. For more information about XLINK options versus ILINK options, see *Differences related to linker options*, page 28. See also *Linker and linker configuration*, page 22.

- **Output Converter** category

  In version 4.x, XLINK can produce a number of output formats and you specify on the linker **Output** page which one to be used. In version 5.x, ILINK produces

ELF/DWARF. Use the **Output Converter** options to convert the ELF output to either Intel-hex or Motorola S-records.

- **Library Builder** category

  In version 5.x, there is a new library builder, which means no options are converted automatically. During the project conversion, all library builder options will be set to default values.

Remember to set any new options.

For information about where to set the equivalent options in the IAR Embedded Workbench IDE, see the *IAR Embedded Workbench® IDE User Guide for ARM®*.

Runtime environment and object files

INTEROPERABILITY

To build code produced by version 5.x of the compiler, you must use the runtime environment components it provides. It is not possible to link object code produced using version 5.x with components provided with version 4.x. This means that you must rebuild your version 4.x object code and in some cases you might need to make some source code modifications.

If you want your application to be AEABI compliant (Embedded Application Binary Interface for the ARM architecture), thus be able to build an application containing object files from different vendors using a third-party linker, you must enable AEABI compliance in the tools. For information about how to do this, see the *IAR C/C++ Development Guide for ARM®*.

SELECTING RUNTIME LIBRARY FILES

In version 4.x, the C/C++ standard library is provided as prebuilt runtime library files including additional support routines, where each file is built using a specific set of compiler options.

Depending on the compiler options you are using for your project, you should specify a runtime library file on the command line that matches your project options. In the IAR Embedded Workbench IDE version 4.x, the correct library file is automatically used based on your project settings.

In version 5.x, different groups of runtime library files are provided, where each group is built using a specific set of compiler options.

Depending on the compiler options you are using for your project, ILINK automatically uses the correct standard library file. If necessary, you can also specify library files

manually directly on the command line or in the IAR Embedded Workbench IDE version 5.x.

To read more about the library files available in version 5.x, see the *IAR C/C++ Development Guide for ARM®*.

Debugger

The directory structure for device description files (`ddf` files) has changed. In version 5.x, these files are placed in device-specific subfolders. If you choose a device description file explicitly, you must pay attention to the new directory structure.

FLASH LOADERS

To use a flash loader for downloading your application, an additional output file in the `simple-code` format is required. In version 4.x, you have to manually set up XLINK to generate this extra `sim` file. In version 5.x, this additional file is not required as C-SPY automatically generates the information for the download.

Tools options

This section lists the differences between the command line options in version 4.x and version 5.x, for the compiler, assembler, and the linker.

DIFFERENCES RELATED TO COMPILER OPTIONS

The following table shows the version 4.x compiler command line options that have changed:

Old compiler option	Description	In version 5.x
`--library_module`	Creates a library module	Removed
`--module_name`	Sets the object module name	Removed
`--omit_types`	Excludes type information	Removed
`--segment`	Changes a segment/section name	`--section`
`--separate_cluster_for_initialized_variables`	Separates initialized and non-initialized variables	Removed
`-s` and `-z`	Sets optimization level	`-O`

Table 4: Differences in compiler options

-s and -z versus -O

In version 5.x, the compiler option for setting the optimization level behaves differently compared to in version 4.x. In version 4.x, there is one option for setting the level of speed optimization and one for setting the level of size optimization, and you choose one of them. In version 5.x, there is an option for setting the level of optimization, and for each level the compiler balances between size and speed. For the highest level it is possible to fine-tune the optimizations explicitly for either size or speed.

The following table shows the mapping between `-s` and `-z` compared to `-O`:

In version 4.x	In version 5.x
`-s2`, `-z2`	`-On` (none)
`-s3`, `-z3`	`-Ol` (low)
`-s6`, `-z6`	`-Om` (medium)
`-z9`	`-Ohz` (high, favoring size)
`-s9`	`-Oh` (high, balancing between speed and size)
--	`-Ohs` (high, favoring speed)

Table 5: Differences in compiler options

Note: In version 5.x, using the options `-s` and `-z` will result in a diagnostic message.

DIFFERENCES RELATED TO ASSEMBLER OPTIONS

The following table shows the version 4.x assembler command line options that have changed:

Old assembler option	Description	In version 5.x
`-b`	Creates a library module	Removed
-X	Unreferenced externals in object files	Removed

Table 6: Differences in assembler options

DIFFERENCES RELATED TO LINKER OPTIONS

The following table summarizes the XLINK command line options and refers you to the corresponding functionality in ILINK:

XLINK option	Description	In ILINK
`-!`	Comment delimiter	In the `icf` file, `/*...*/` or `//`.
`-A`	Loads as program	Removed, see *Assembler and assembler source code*, page 20.

Table 7: Counterparts to XLINK options in ILINK

XLINK option	Description	In ILINK
-a	Disables static overlay	Removed
-B	Always generates output	--force_output
-b	Defines banked segments	In the icf file *
-C	Loads as a library	Removed, see *Assembler and assembler source code*, page 20.
-c	Specifies the processor type	Removed
-D	Defines a symbol	--define_symbol
-d	Disables code generation	Removed
-E	Inherent, no object code	Removed
-e	Renames external symbols	--redirect
-F	Specifies the output format	Removed
-f	Specifies the XCL filename	-f; in ILINK the configuration file is specified using the option --config
-G	Disables global type checking	Removed
-g	Requires global entries	--keep
-H	Fills unused code memory	--fill
-h	Fills ranges	--fill
-I	Specifies the include paths	Removed
-J	Generates a checksum	--checksum
-K	Duplicates code	In the icf file *
-L	Lists to directory	--log_file
-l	Lists to a named file	--log_file
-M	Maps logical addresses to physical addresses	In the icf file *
-n	Ignores local symbols	--no_locals
-O	Multiple output files	Removed
-o	Output file	Unchanged, but --output can also be used as an alias.
-P	Defines packed segments	In the icf file *
-p	Specifies lines/page	Removed
-Q	Scatter loading	In the icf file *
-q	Disables relay function optimization	Removed

Table 7: Counterparts to XLINK options in ILINK (Continued)

XLINK option	Description	In ILINK
-R	Disables range check	--diag_suppress
-r	Debug information	Removed. In ILINK, debug information is included by default, and removed by using --no_debug.
-rt	Debug information with terminal I/O	--semihosting
-S	Silent operation	--silent
-s	Specifies a new application entry point	--entry
-U	Address space sharing	In the icf file *
-V	Declares relocation areas for code and data	In the icf file *
-w	Sets diagnostics control	--diag_error, --diag_remark, --diag_suppress, --diag_warning, --diagnostics_tables, --error_list, --no_warnings, --remarks, --warnings_are_errors, --warnings_affect_exit_code
-x	Specifies cross-reference	--map
-Y	Format variant	Removed
-y	Format variant	Removed
-Z	Defines segments	In the icf file *
-z	Segment overlap warnings	Removed

Table 7: Counterparts to XLINK options in ILINK (Continued)

* In ILINK, this functionality is not available as a linker option that you specify either on the command line or in the IAR Embedded Workbench IDE. Instead, it is part of the configuration that you specify in the linker configuration file.

The following options have not changed in any significant way:

--image_input, --misrac, --misrac_verbose.

Segments versus sections

This part describes the differences between segments in version 4.x and sections in version 5.x.

For detailed information about the new sections, their names, and how they are used, see the *IAR C/C++ Development Guide for ARM®*.

NAMING CONVENTIONS

The naming convention for segments differs slightly compared to the naming convention for sections.

In 4.x, all segment names are written with capital letters. For the code and data segments, the segment base name indicates the memory type. In addition, the static data segments also have a suffix indicating type of contents.

In 5.x, some sections are written with capital letters and some with lower-case letters with a preceding period. For these sections, the section name indicates the type of contents and the suffix indicates the memory type.

SEGMENTS FOR INITIALIZATION

In 4.x, the compiler creates one segment for initializers and one segment for the initialized variables. In 5.x, the compiler creates one data section that contains the initializers. ILINK then transforms that section into proper handling of the initialization. To read more about initializations, see the *IAR C/C++ Development Guide for ARM®*.

MAPPING OLD SEGMENTS TO NEW SECTIONS

Some of the old segments have disappeared entirely, and some of the new sections do not have any counterparts among the old segments.

This table lists the old segment names, their counterparts in version 5.x, and additional sections:

Old segment	New section	Comments
`CODE`	`.text`	
`CODE_I`	`.textrw`	
`CODE_ID`	`.textrw`	Initializers no longer have a section of their own.
`CSTACK`	`CSTACK`	

Table 8: Mapping segments and sections

Old segment	New section	Comments
DATA_AC	--	Absolute placement of constants is no longer supported, which means a dedicated segment/section is no longer needed.
DATA_AN	--	Absolute __no_init declared variables no longer reserves space, which means a dedicated segment/section is no longer needed.
DATA_C	.rodata	
DATA_I	.data	
DATA_ID	.data	Initializers no longer have a section of their own.
DATA_N	.noinit	
DATA_Z	.bss	
DIFUNCT	.difunct, PREDIFUNCT	
HEAP	HEAP	
ICODE	.text	
INITTAB	--	ILINK uses a different method to solve this, which means a dedicated segment/section is no longer needed.
INTVEC	.intvec	
IRQ_STACK	IRQ_STACK	
SWITAB	--	This functionality has been removed, which means a dedicated segment/section is no longer needed.

Table 8: Mapping segments and sections (Continued)

Assembler directives

Some of the assembler directives have been removed or behave differently. The following table lists the assembler directives which are not the same in version 5.x as in version 4.x:

Assembler directives in version 4.x	In version 5.x
ARGFRAME	Removed

Table 9: Differences in assembler directives

Assembler directives in version 4.x	In version 5.x
ASEG	Removed
ASEGN	Removed
BLOCK	Removed
CFI	The resource names are standardized. A CFI names block must contain a subset of these resource names.
COMMON	Removed
DEFFN	Removed
END	No longer takes a program start address as an argument.
ENDMOD	Recognized but without effect; a warning is generated.
FUNCALL	Removed
FUNCTION	Removed
LIBRARY	Instead of starting a library module, it now starts an ELF module. New syntax.
LIMIT	Removed
LOCFRAME	Removed
MODULE	Instead of starting a library module, it now starts an ELF module. New syntax.
MULTWEAK	Removed
NAME	Starts an ELF program module. New syntax.
ORG	Removed
OVERLOAD	Removed
PROGRAM	Starts an ELF program module. New syntax.
RSEG	The first instance of the RSEG directive used must not be preceded by any code generating directives, such as DC or DS, or by any assembler instructions. This directive is now an alias for the new directive SECTION. New syntax.
STACK	Removed
SYMBOL	Removed

Table 9: Differences in assembler directives (Continued)

For information about assembler directives in version 5.x, see the *ARM® IAR Assembler Reference Guide*.

Filename extensions

The following table lists the differences related to default filename extensions:

Old filename extension	New filename extension	Description/Comments
`s79`	`s`	Assembler source file
`r79`	`o`	Object module
`r79`	`a`	Library object module
`a79`	`out`	Target program
`d79`	`out`	Target program for debugging

Table 10: Differences in filename extensions

Migrating to ARM® IAR Embedded Workbench version 4.x

This guide gives hints for porting your application code and projects to version 4.x.

Code that was originally written for the ARM IAR C/C++ Compiler version 3.x can also be used with ARM IAR C/C++ Compiler version 4.x, although some modifications may be required.

This guide first presents the major differences between the two product versions and then gives an overview of the migration process. Finally, it describes the differences between the ARM IAR Embedded Workbench version 3.x and version 4.x. Both differences and similarities between the products are briefly discussed.

Key advantages

This section lists the major advantages in the ARM IAR Embedded Workbench version 3.x as compared to the ARM IAR Embedded Workbench version 4.x. Hereafter, the two versions are referred to as *version 3.x* and *version 4.x*, respectively.

- Efficient window management through dockable windows optionally organized in tab groups
- Source browser with a catalog of functions, classes, and variables, for a quick navigation to symbol definitions
- Template projects to get a project that links and runs *out of the box* for a smooth development start-up
- Batch build with ordered lists of configurations to build
- Improved context-sensitive help for C/C++ library functions
- Generic flash downloader framework
- Improved compiler optimizations with up to 10% smaller code
- Support for ARM Embedded Trace Macrocell using the EPI Majic JTAG interface

- New coprocessor intrinsics
- Easy configuration of the C/C++ libraries
- New keywords with support for nested interrupts
- Smart display of STL containers at debugging
- Auto-display debugger watch window
- A broad range of feature enhancements.

Migration considerations

To migrate your old project consider the following:

- IAR Embedded Workbench IDE
- Code models and code generation
- Project options
- 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 3.x might generate warnings or errors in version 4.x.

IAR Embedded Workbench IDE

Upgrading to the new version of the IAR Embedded Workbench IDE should be a smooth process as the improvements do not have any major influence on the compatibility between the versions.

WORKSPACE AND PROJECTS

The workpaces and projects you have created with 3.x are compatible with version 4.x. Note that there are some differences in the project settings. Therefore, make sure to check the options carefully. For further information, see *Project options*, page 39.

C-SPY LAYOUT FILES

Because of new improved window management system, the C-SPY layout files support in 3.x has been removed. Any custom made `lew` files can safely be removed from your projects.

Code models and code generation

In version 3.x, you could choose between two different code models in the ARM IAR Embedded Workbench. The small code model was limited to 4 Mbytes of code in Thumb mode and 32 Mbytes in Arm mode, whereas the large code model allowed unlimited use of the entire addressable memory space in the Arm core. However, even in the large code model, the individual linker segments were limited to 4 and 32 Mbytes for Thumb and Arm mode respectively.

The placement of code into specifically named segments was based on a number of factors, such as used code model, function memory attributes, pragma segment directives, and the `--segment` command line option.

Due to improvements in the compiler, and particularly the linker, the split into two code models has become obsolete. This change also affects:

- Segments
- Extended keywords
- Pragma directives.

SEGMENTS

Segment parts containing Thumb and Arm code are now placed side by side in the same segments. Some of the advantages are: automatic generation of more compact code, unlimited segment sizes, and a simplified configuration process.

As a direct consequence, the segment naming has been changed. The linker command files shipped with version 4.x have been modified accordingly, but if you use a customized linker command file, you also need to modify your linker command file accordingly. Note that the name changes affect data segments as well as code segments.

Old and new segments

The following table summarizes the segment transition needed when migrating from an earlier version of IAR Embedded Workbench than version 4.x:

Old segment	New segment
FARFUNC_A	CODE
FARFUNC_T	CODE
NEARFUNC_A	CODE
NEARFUNC_T	CODE
FARFUNC_A_I	CODE_I

Table 11: Old and new segments

Old segment	New segment
FARFUNC_T_I	CODE_I
NEARFUNC_A_I	CODE_I
NEARFUNC_T_I	CODE_I
FARFUNC_A_ID	CODE_ID
FARFUNC_T_ID	CODE_ID
NEARFUNC_A_ID	CODE_ID
NEARFUNC_T_ID	CODE_ID
HUGE_AC	DATA_AC
HUGE_AN	DATA_AN
HUGE_C	DATA_C
HUGE_I	DATA_I
HUGE_ID	DATA_ID
HUGE_N	DATA_N
HUGE_Z	DATA_Z

Table 11: Old and new segments (Continued)

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

EXTENDED KEYWORDS

As there is no longer a choice of code or data models, all function and data memory attributes have become obsolete. The compiler will therefore issue a warning message for any of the following obsolete keywords:

```
__farfunc, __nearfunc, __huge
```

PRAGMA DIRECTIVES

The removal of all function and data memory attributes has implicated a change to the pragma directive

```
#pragma segment="segment"
```

For information about the new syntax, see *ARM® IAR C/C++ Compiler Reference Guide.*

Project options

In version 4.x, there are several new project options. For information about the command line variants, see the *ARM® IAR C/C++ Compiler Reference Guide.* For information about the IAR Embedded Workbench variants, see the *IAR Embedded Workbench® IDE User Guide for ARM®*.

When migrating from an earlier version of the ARM IAR Embedded Workbench, the following changes to the project options are particularly important:

Command line	IAR Embedded Workbench	Description
`--code_model`	Code model	Obsolete
`--segment`	Segment names	Changed syntax
`--no_tbaa`	Type-based alias analysis	New optimization, enabled by default

Table 12: Project options

As there is no longer a choice of code or data models, the `--code_model` option has become obsolete. The compiler will therefore issue a warning message when encountering this option. For more information, see *Code models and code generation*, page 37.

The syntax of the option `--segment` has changed. The compiler will issue a warning message when the old syntax is used.

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.

Runtime library and object files considerations

To build code produced by version 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 4.x with components provided with version 3.x.

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 ARM IAR Embedded Workbench version 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. 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 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, the same library configuration file must be used as when the library was built. For the prebuilt libraries (`r79`) there is a corresponding library configuration file (`h`), which has the same name as the library. The files are located in `arm\lib`. The command lines for specifying the library configuration file and library object file could look like this:

```
iccarm -D_DLIB_CONFIG_FILE=...\arm\lib dl4tpainl8n.h
xlink dl4tpainl8n.r79
```

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

To take advantage of the features it is recommended that you read about the runtime environment in the *ARM® IAR C/C++ Compiler 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 4.x, the default program entry label in `cstartup.s79` 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.s79`.

If you build your application in 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.s79` file has been split up into three files:

```
cstartup.s79, cmain.s79, cexit.s79
```

Now, the `cstartup.s79` only contains exception vectors and initial startup code to setup stacks and processor mode. Note that the `cstartup.s79` file is the only one of these three files that may require any modifications.

The `cmain.s79` file initializes data segments and executes C++ constructors. The `cexit.s79` file contains termination code, for example, execution of C++ destructors.

For applications that use a modified copy of `cstartup.s79`, you must adapt it to the new file structure.

CSTARTUP RESET VECTOR

Before version 3.x, the reset vector in `cstartup.s79` was implemented as a relative branch to the first instruction at `?cstartup`

```
B ?cstartup
```

The drawback of this approach was that the branch range with the `B` instruction is limited to 32 Mbyte. It is fairly common, especially during debugging, to place the code in a memory in the upper part of the address map, which means the code is located beyond the reach of the `B` branch instruction.

To remove this limitation, 3.x was changed so that an absolute jump is made instead of a branch instruction:

```
LDR PC,=?cstartup
```

To accommodate for the extra constant value, the reserved vector area was increased from 32 bytes to 64 bytes. With this solution, the reset vector can reach anywhere within the full 4 Gbyte address space. This usually works fine when debugging an application.

However, there are two situations when the solution can cause problems:

- When downloading a program to flash memory located at zero that will later be remapped to another location
- If the code is copied and executed in RAM.

For an application with the reset vector at zero and code that will be relocated to the address `0x100000`, the application will be linked with the assumption that it is located at that address. At reset, the content of address `0x100000` will be seen at address zero.

Because the reset vector contains an absolute jump, for example to `0x100100`, the jump will fail, because the code is not yet mapped or copied to that address. In this situation, a relative branch instruction would work.

If your application uses the remap or copy mechanism, you should modify the `cstartup.s79` file and change the reset vector to do a relative jump instead of the default absolute jump.

DEVICE SPECIFIC HEADER FILES

Some of the header files defining peripheral registers have changed their register naming convention. The double underscores __ preceeding the register name have been removed. A copy of the old style file is delivered with the product for backward compatibility. For example, the old version of `iolpc210x.h` is now available as `iolpc210x_old.h`.