MAXQ IAR Embedded Workbench®

Migration Guide

for Dallas Semiconductor/Maxim's

MAXQ Microcontroller

EDITION NOTICE

First edition: March 2006

Part number: MMAXQ-1

This guide applies to version 2.x of MAXQ IAR Embedded Workbench®.

Contents

Migrating from version 1.xx to version 2.xx

This chapter gives hints for porting your application code and projects to the new version 2.xx.

C source code that was originally written for the MAXQ IAR C Compiler version 1.xx can be used also with the new MAXQ IAR C Compiler version 2.xx. However, some small modifications may be required.

This guide presents the major differences between the MAXQ IAR Embedded Workbench version 1.xx and the MAXQ IAR Embedded Workbench version 2.xx, and describes the migration considerations.

The migration process

In short, to migrate your old project consider the following:

- Some minor changes in the IAR Embedded Workbench IDE
- Project setup related to the new code and data models
- 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.

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 affect the compatibility between the versions.

WORKSPACE AND PROJECTS

The workspaces and projects you have created with 1.xx are compatible with 2.xx. Note that there are some differences in the project settings. Therefore, make sure to check the options carefully. For further information, see *Code and data models*, page 2.

C-SPY® LAYOUT FILES

Due to a new improved window management system, the C-SPY layout files support in 1.xx has been removed. Any custom-made `lew` files can safely be removed from your projects.

Code and data models

In version 2.xx, two code models and two data models have been introduced. These are mechanisms for flexible utilization of code and data memory, and for some applications can lead to smaller code size. These models are also good for portability reasons as no target-specific keywords are needed.

In short about the code models:

- The *Small* code model is segmented and suitable for devices with less than 64 Kbytes program memory or applications with lots of constant data
- The *Large* code model is linear and suitable for larger devices and for applications that can have its constant data in RAM. This code model is only available for the MAXQ20 core.

Use the `--code_model={small|large}` option to choose code model for your project.

In short about the data models:

- The *Small* data model can access the low 256 bytes of the data memory space
- The *Large* data model can access the entire 64 Kbytes of the data memory space.

Use the `--data_model={small|large}` option to choose data model for your project.

For more detailed information, see the *MAXQ IAR C Compiler Reference Guide*.

LINKER CONSIDERATIONS

If you have created your own customized linker command file, compare this file with the original file in the old installation and make the required changes in a copy of the corresponding file in the new installation. Note that the new code and data models also introduces some changes among the linker segments.

Linker segments for holding data

All segment names previously named `DATA_`*`suffix`* have changed names to `DATA16_`*`suffix`*. These segments are used by default to hold data when compiling using the Large data model, and for data that is explicitly declared `__data16`. In addition, a new set of segments, `DATA8_`*`suffix`*, have been added for holding data when compiling using the Small data model and for data explicitly declared `__data8`.

Linker segments for holding code

In version 2.xx, the following code segments are new or have modified behavior:

- The `CODE` segment holds `__near_func` program code and constant data when compiling using the `--place_const_in_code` option
- The `LCODE` segment holds program code in the Large code model.

The segments `FARCODE` and `RCODE` have the same behavior as in version 1.xx.

Runtime library and object files considerations

In version 2.xx, two sets of runtime libraries are provided—CLIB and DLIB. CLIB corresponds to the runtime library provided with version 1.xx, and it can be used in the same way as before. However, for changes related to DLIB, see *Compiling and linking with the DLIB runtime library*, page 3.

To build code produced by version 2.xx of the compiler, you must use the runtime environment components it provides. It is not possible to link object code produced using version 2.xx with components provided with version 1.xx.

For information about how to migrate from CLIB to DLIB, see *Migrating from CLIB to DLIB*, page 5. For more information about the two libraries, and the runtime environment they provide see the *MAXQ IAR C Compiler Reference Guide*.

COMPILING AND LINKING WITH THE DLIB RUNTIME LIBRARY

In earlier versions, the choice of runtime library did not have any impact on the compilation. In MAXQ IAR Embedded Workbench version 2.xx, 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 might use the `fprintf` function for terminal I/O (`stdout`), but might 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 (`r66`) there is a corresponding library configuration file (`h`), which has the same name as the library. The files are located in the `maxq\lib` directory. The command lines for specifying the library configuration file and library object file could look like this:

```
iccmaxq -dlib_config <install_dir>\maxq\lib\dlib
dlmaxq10fsln.h

xlink dlmaxq10fsln.r66
```

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

To take advantage of the new features, it is recommended that you read about the runtime environment in the *MAXQ IAR 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.

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 2.xx, the default program entry label in `cstartup.s66` 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.s66`.

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 application. If you link without the option, the resulting output executable file will be empty, because no modules were referred to.

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 1.xx 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 *MAXQ IAR C Compiler Reference Guide*.