8051 IAR Embedded Workbench

Migration Guide

for the

MCS-51 Microcontroller Family

EDITION NOTICE

Second edition: July 2005

Part number: M8051-2

This guide applies to version 7.x of 8051 IAR Embedded Workbench®.

Contents

Tables

Migrating from version 6.x to version 7.10A

This guide gives hints for porting your application code and projects to the new version 7.10A.

C source code that was originally written for the 8051 IAR C Compiler version 6.x can be used also with the new 8051 IAR C/C++ Compiler version 7.10A. However, some small modifications may be required.

This guide presents the major differences between the 8051 IAR Embedded Workbench version 6.x and the 8051 IAR Embedded Workbench version 7.10A, and describes the migration considerations.

Note that if you are migrating from the 8051 IAR Embedded Workbench version 5.x , you must first read the chapter *Migrating from version 5.x to version 6.x*.

Key advantages

This section lists the major advantages in the 8051 IAR Embedded Workbench version 6.x as compared to the 8051 IAR Embedded Workbench version 7.10A. Hereafter, the two versions are referred to as *version 6.x* and *version 7.10A*, 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
- Smart display of STL containers at debugging
- Auto-display debugger watch window
- Support for the C++ language
- Support for the DLIB runtime environment
- A broad range of small feature enhancements.

Migration considerations

To migrate your old project consider changes related to:

- IAR Embedded Workbench IDE
- Project options
- Runtime library and object files.

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 6.x may generate warnings or errors in version 7.10A.

IAR Embedded Workbench IDE

Upgrading to the new version of the IAR Embedded Workbench IDE should be a smooth process as the improvements hot have any major influence on the compatibility between the versions.

WORKSPACE AND PROJECTS

The workpaces and projects you have created with 6.x are compatible with 7.10A. 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 2.

C-SPY LAYOUT FILES

Due to new improved window management system, the C-SPY layout files support in 6.x has been removed. Any custom made `lew` files can safely be removed from your projects.

Project options

In version 7.10A, there are several new project options, for instance for the new support for the DLIB runtime environment, MISRA C, and C++. Note also that the option layout and syntax has been changed. For information about the command line variants, see the *8051 IAR C/C++ Compiler Reference Guide.* For information about the IAR Embedded Workbench variants, see the *IAR Embedded Workbench® IDE User Guide*.

Runtime library and object files considerations

In version 7.10A, two sets of runtime libraries are provided—CLIB and DLIB. CLIB corresponds to the runtime library provided with version 6.x, and it can be used in the same way as before.

To build code produced by version 7.10A of the compiler, you must use the runtime environment components it provides. It is not possible to link object code produced using version 7.10A with components provided with version 6.x.

For information about how to migrate from CLIB to DLIB, see *Migrating from CLIB to DLIB*, page 4. For more information about the two libraries, and the runtime environment they provide see the *8051 IAR C/C++ Compiler Reference Guide*.

COMPILING AND LINKING WITH THE DLIB RUNTIME LIBRARY

With DLIB you can configure the runtime library to contain the features that are needed by your application.

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

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

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

When building an application from the command line, you must use the same library configuration file as when the library was built. For the prebuilt libraries (`r51`) there is a corresponding library configuration file (`h`), which has the same name as the library. The files are located in the `8051\lib` directory. The command lines for specifying the library configuration file and library object file could look like this:

```
icc8051 --dlib_config ...\8051\lib dl8051Normal.h
xlink dl-ele-ffxd-2e24inc.r51
```

In case you intend to build your own library version, use the default library configuration file `dl8051Custom.h`.

To take advantage of the features it is recommended that you read about the runtime environment in the *8051 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 7.10A, the default program entry label in `cstartup.s51` 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.s51`.

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

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

SYSTEM INITIALIZATION—CSTARTUP

The content of the `cstartup.s51` file has been split up into three files:

```
cstartup.s51, cmain.s51, cexit.s51
```

Now, the `cstartup.s51` only contains exception vectors and initial startup code to setup stacks and processor mode. Note that the `cstartup.s51` file is the only one of these three files that may require any modifications.

The `cmain.s51` file initializes data segments and executes C++ constructors. The `cexit.s51` file contains termination code, for example, execution of C++ destructors.

For applications that use a modified copy of `cstartup.s51`, 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 6.x project using CLIB was defined in a file named `heap.c`, you must now set the heap size either in the extended linker command file (`*.xcl`) or in the Embedded Workbench to use the DLIB library.

You should also see the chapter *The DLIB runtime environment* in the *8051 IAR C/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 many of the segment names have changed, see the chapter *Segments* in the *8051 IAR C/C++ Compiler Reference Guide*.

Segment control directive -b vs -P

If you were using the linker segment control directive `-b` for locating your banked code in memory, be aware that this directive is now obsolete and superseded by the new linker segment control directive `-P`. For details of the `-P` directive, see the *IAR Linker and Library Tools Reference Guide*.

Migrating from version 5.x to version 6.x

This guide gives hints for porting your application code and projects to the 8051 IAR Embedded Workbench IDE version 6.x.

C source code that was originally written for the 8051 IAR C Compiler version 5.x can be used also with the new 8051 C/EC++ Compiler version 6.x. However, some modifications may be required.

This guide contains information about migrating from 8051 IAR Embedded Workbench 5.x to version 6.x. It presents the major differences between 8051 IAR Embedded Workbench version 5.x and 8051 IAR Embedded Workbench version 6.x, and describes the migration considerations.

Key advantages

This section lists the major advantages in the 8051 IAR Embedded Workbench version 6.x as compared to version 5.x.

- Improved compiler optimizations
- Highly optimized reentrant code generation
- Pre-configured device support
- Full support for multiple DPTRs
- More flexible runtime models
- Auto-display and live debugger watch window
- Easy configuration of the C/C++ libraries
- A broad range of feature enhancements.

IDE

- Batch build with ordered lists of configurations to build
- Total integration of compiler and debugger toolkit under a modular and extensible IDE
- Full support for both classic and extended 8051 architectures, like those from Maxim/Dallas Semiconductor and Analog Devices
- Multiple projects in the same workspace

- Hierarchical project representation shows all different source and output files and gives an overview of their settings
- XML-based project files
- Multibyte editor
- Easy to integrate external tools in the build process
- Preconfigured device support for most derivatives on the 8051 market.

COMPILER

- Highly optimized ISO/ANSI standard C compiler generating the most compact code on the market
- Multiple levels of both size and speed optimizations
- Easy and fast interrupt handling directly in C
- User control of register usage for optimal performance
- Support for multiple DPTRs in compiler and libraries
- Local bit and bit parameters
- Possibility to use up to 32 virtual registers.

DEBUGGER

- Complex code and data breakpoints with resume functionality
- Function call stack window
- Support for stack unwinding even at high optimization levels
- Fine-grain single stepping on a function call level
- Terminal I/O, peripheral and interrupt simulation
- Memory configuration and validation
- Versatile monitoring of CPU/peripherals, registers, structures, call chain, local and global variables
- RTOS-aware debugging
- Window for viewing stack contents.

Migration considerations

In short, to migrate to the new version, pay attention to the following:

- Project file and project setup
- Source code and compiler considerations
- Assembler considerations
- Runtime environment considerations
- Linker considerations

Moreover, version 6.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 also adheres more strictly to the ISO/ANSI C standard, compared to version 5.x.

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

Project file and project setup

To migrate your old project, follow the described migration process. Note that not all steps in the process might be relevant for your project. Consider carefully what actions are needed in your case.

Project files created in version 5.x and version 6.x of 8051 IAR Embedded Workbench are not compatible. Therefore a version 5.x project file cannot be converted to a version 6.x project file. A new workspace and project must be created for a version 6.x project. Follow these steps:

1. Start your new version of 8051 IAR Embedded Workbench and create a new workspace by choosing **File>New** and then **Workspace**.
2. Choose **Project>Create New Project** to create a new project in the workspace. For more information about creating and setting up projects, see the *IAR Embedded Workbench® IDE User Guide.*
3. Make sure you have a backup of your old project and manually add all files from the version 5.x project to the new project.
4. Select the appropriate runtime model, see *Runtime environment considerations*, page 14, and *Memory models*, page 11.
5. Set the appropriate compiler options, see *Compiler options*, page 21. To generate a text file with the command line equivalents of the project options in your old project, see *Migrating project options*, page 21.
6. Make sure to use the header files delivered with version 6.x, since the header files that define peripheral registers delivered with version 5.x cannot be used with version 6.x.
7. Convert the version 5.x source code to code that is accepted by version 6.x of the compiler and the assembler, see *Source code and compiler considerations*, page 10.
8. Make a copy of the linker command file—`lnk51`*`n`*`.xcl`, supplied with version 6.x—that matches your chip best. Compare it with the file used to link the version 5.x project, and make the required changes. Do not use your old linker command file, and do not make any changes in `lnk_base.xcl`. For information about changes related to segments, see *Segments*, page 33.

The specification of the runtime library has, together with the possibility to ignore `cstartup` in the library, been moved from the linker command file. This means that it is now possible to use the same linker command file for both C/EC++ and assembler code. The **Ignore CSTARTUP in library** option can be set on the **Include options** page in the **XLINK** category.

Source code and compiler considerations

In short, the process of migrating from version 5.x to version 6.x involves the following steps:

1. Decide which data model, code model and calling convention to use, see *Memory models*, page 11.
2. Find all generic pointers in the version 5.x application that are used to access memory not reachable by the version 6.x default pointer, see *Generic pointers*, page 16.
3. Replace or modify extended keywords according to the description in the section *Extended keywords*, page 25.
4. Replace or modify pragma directives according to the section *Pragma directives*, page 29.
5. Make sure not to use nested comments in your source code. In version 6.x, nested comments are never allowed.
6. Replace or modify intrinsic functions according to the section *Intrinsic functions*, page 32.

Note: Version 6.x will by default not accept preprocessor expressions containing any of the following:

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

With the option `--migration_preprocessor_extensions`, version 6.x will accept such non-standard expressions. For details about this option, see the *8051 IAR C/C++ Compiler Reference Guide.*

To set the equivalent option in the IAR Embedded Workbench IDE choose **Project>Options>ICC8051>Language**.

PREDEFINED SYMBOLS

All predefined symbols supported in version 5.x of the compiler are also supported in version 6.x. Note that the semantics for the `__TID__` has changed in version 6.x.

Version 6.x does not support the concept of memory models used in version 5.x; the correspondence is instead what calling convention as well as data and code models are used. There are also additional predefined symbols in version 6.x.

See the chapter *Predefined symbols* in the *8051 IAR C/C++ Compiler Reference Guide* for information about the predefined symbols available in version 6.x.

MEMORY MODELS

Version 5.x of the compiler uses memory models to select the default location for local and global data. This is not the case in version 6.x, which instead uses the concepts of *data model*, *code model* and *calling convention*. The data model affects where global data is stored and also which default pointer is used. The code model specifies in which memory the functions are located. The calling convention affects where local data and parameters are stored. The following table can be used when converting an old version 5.x memory model to the new version 6.x data model, code model and calling convention.

Note: There are more combinations of these options available than the ones listed in the table; the table only lists a few possible combinations as a guideline.

Memory model (version 5.x)	Data model (version 6.x)	Code model (version 6.x)	Calling convention (version 6.x)
Tiny	Tiny	Tiny/near	Data overlay, Idata overlay, Idata reentrant
Small	Small	Near	Idata overlay, Idata reentrant
Compact	Large	Near	Pdata reentrant
Medium	Large	Near	Pdata reentrant, Xdata reentrant
Large	Large, (Generic)	Near	Xdata reentrant
Banked	Large, (Generic)	Banked	Xdata reentrant

Table 1: Converting from version 5.x memory models

The compact and medium memory models in version 5.x do not have a similar counterpart in version 6.x. However, the option combinations listed in the table might be the best choice.

The memory model option in version 5.x of the compiler affects several different parameters that have now been separated into different options. This makes version 6.x much more flexible. However, not all combinations of the data models and calling conventions in version 6.x can be combined with each other. The reason for this is that the default pointer depends on the selected data model and must be able to reach locally defined variables. The location for these variables depends on the selected calling convention. For more information about the data model, code model and calling convention in version 6.x and about how these can be combined, see *Customization* in the *8051 IAR C/C++ Compiler Reference Guide*.

By default all constants are placed in data memory in version 6.x. Version 6.x of the compiler offers the same support regarding the location of constants and strings as in version 5.x, see *Constants and strings*, page 17.

Note: Even though version 6.x of the compiler supports a data model that uses the generic pointer as the default pointer, we do not recommend using this because it does not generate optimized code.

In 8051 IAR Embedded Workbench there are tools for source code conversion available from the **Tools** menu. One of the tools can convert large parts of your old C source code to the new version 6.x syntax.The tool comments each statement that has to be modified in order to comply with the new syntax.

Assembler considerations

Version 6.x of the compiler uses a completely different calling convention than version 5.x. Any C code that interfaces with assembler routines and vice versa must be adapted to the new calling convention. C code calling an assembler routine must be changed so that the correct registers are used for parameters and return values. An assembler routine called from C must be changed so that arguments and return values passed to and from it are located in the registers that adhere to the version 6.x calling convention. For more information on the calling convention used in version 6.x, see *Assembler language interface* in the *8051 IAR C/C++ Compiler Reference Guide*.

BYTE ORDER

The byte order has changed between version 5.x and version 6.x. Version 6.x adheres to little endian byte order. The static overlay mechanism has also changed from version 5.x to version 6.x. Thus all old version 5.x operators used for handling static overlay have been removed and replaced by new operators.

REMOVED OPERATORS

The following operators have been removed:

- $BYTE3
- $PRMBB
- $PRMBD
- $PRMBI
- $PRMBX

- $LOCBB
- $LOCBD
- $LOCBI
- $LOCBX
- $REFFN
- $IFREF
- $REFFNT

NEW OPERATORS

The following operators have been added:

- PRM
- LOC
- BYTE1
- BYTE4

REMOVED DIRECTIVES

The following directives have been removed:

- CYCLES
- CYCMAX
- CYCMEAN
- CYCMIN
- LSTWID
- LSTFOR
- TITL
- HEADER
- PTITL
- STITL
- PSTITL
- $deffn

NEW DIRECTIVES

The following directives have been added:

- REQUIRE
- CFI
- RTMODEL
- ODD
- PUBWEAK
- FUNCTION
- ARGFRAME
- LOCFRAME

- FUNCALL
- DC8
- DC16
- DC24
- DC32
- DS8
- DS16
- DS24
- DS32

For more information about the new assembler functionality, see the *8051 IAR Assembler Reference Guide.*

Runtime environment considerations

To build an application produced by version 6.x of the compiler, you must use the runtime environment components it provides and rebuild your projects. It is not possible to link object code produced using version 6.x with components provided with version 5.x, see *Linker considerations*, page 19.

RUNTIME LIBRARY

A newer version of the IAR CLIB library is used together with the 8051 IAR C/C++ Compiler version 6.x. The old source code for the IAR CLIB library distributed with version 5.x of the compiler cannot be used together with version 6.x. The complete source code for the IAR CLIB library is delivered with the product. When linking your application, use a runtime library that matches the runtime options selected for the compiled application.

Version 6.x uses a new naming convention for runtime library names. For information about this naming convention, see *Runtime library* in the *8051 IAR C/C++ Compiler Reference Guide*.

There is a great number of runtime options available in the 8051 IAR C/C++ Compiler. Therefore the number of possible runtime libraries is huge. It is only feasible to deliver a few of all possible libraries with the product. For a list of all delivered libraries see *Combinations and dependencies* in the *8051 IAR C/C++ Compiler Reference Guide*.

If your application uses a set of runtime options for which no runtime library is delivered, you have to compile your own library from the library source code. Any changes made to the assembler- or C-written library modules distributed together with version 5.x of the compiler must be migrated to the corresponding module in the CLIB library, distributed with version 6.x. For more information about how to compile your own library see *Building your own runtime library* in the *IAR Embedded Workbench® IDE User Guide*, and *Building a runtime library* in the *8051 IAR C/C++ Compiler Reference Guide*.

For more information about the library, see *Library functions* in the *8051 IAR C/C++ Compiler Reference Guide*. For information about the runtime environment, see *Runtime environment* in the same guide.

Dynamic memory allocation—the heap

The interface to the runtime library functions used to allocate and deallocate memory on the heap has changed. This affects the functions `malloc`, `realloc`, `calloc`, and `free`. As in version 5.x, a heap is only supported in external data memory; if the microcontroller does not have external memory, no heap can be used. For more information about the new interface see *Alternative memory allocation functions* in the chapter *Data storage* in the *8051 IAR C/C++ Compiler Reference Guide*.

When converting a version 5.x application to version 6.x it is recommended that all calls to the functions `malloc`, `calloc`, `realloc`, and `free` are changed to the corresponding functions `__xdata_malloc`, `__xdata_calloc`, `__xdata_realloc`, and `__xdata_free`.

Constants and strings in code memory

If you want to locate constants and strings in code memory, the runtime library must be compiled with the option `--place_constants=code`. Unfortunately, the unmodified runtime library only compiles correctly if the generic pointer is used as the default pointer (selected by using the data model *generic*). Since your application must use the same runtime options as the runtime library the whole application must use generic pointers. This is undesirable since the 8051 IAR C/C++ Compiler is not optimized for generic pointers. Another solution is to modify the source code for the runtime library so that pointers accessing code memory are explicitly declared `__code` or `__generic` and pointers that need to access any data regardless of in which memory it is located, are declared `__generic`. In this case constants and strings can be located in code memory without the need for using the generic pointer as the default pointer. To modify the library source so that constants are located in code memory, compile the library project and explicitly type the pointers that generate compiler errors.

For more information about constants and strings, see *Constants and strings*, page 17.

CALLING CONVENTIONS

The 8051 IAR C/C++ Compiler version 6.x does not support the calling conventions used by version 5.x of the compiler.

Local variables and parameters are handled differently in version 6.x compared to version 5.x. It is no longer possible to explicitly locate local variables and parameters. Instead version 6.x supports six different calling conventions. The selected calling convention affects the location of local data and parameters. The available calling conventions are:

- Data overlay
- Idata overlay
- Idata reentrant
- Pdata reentrant
- Xdata reentrant
- Extended stack reentrant

The overlay calling conventions use a static overlay frame for local data and parameters. The overlay frame will be located in data memory for data overlay functions and in idata memory for idata overlay functions. Note that all local variables and parameters will be located in this memory. The static overlay calling convention is no longer supported for xdata memory. The overlay calling convention does not support recursive or reentrant functions; these functions must use one of the reentrant calling conventions. The reentrant calling conventions use a stack in the respective memory; all local data and parameters will be located on this stack. In addition, the return address is saved on the respective stacks. For more information about the calling conventions in version 6.x, see *Calling convention* in the *8051 IAR C/C++ Compiler Reference Guide*.

GENERIC POINTERS

In version 5.x of the compiler generic pointers were used as the default pointer in all memory models. This is not the case in version 6.x—which is indeed one of the most important changes between version 5.x and version 6.x. Even though generic pointers are supported in version 6.x we recommend against using them as the default pointer, since the resulting code will be unnecessarily large and slow. However, it can be a good idea to use a few explicitly typed generic pointers in an application. When converting a version 5.x application to version 6.x, generic pointers can be troublesome to convert to a more suitable pointer type. A version 5.x application using the more general generic pointer should be rewritten to use more restrictive pointers. Generic pointers are only needed if the same pointer must be able to access data located in different memory types, i.e. internal data memory, external data memory and code memory. This is not a recommended programming practice; it is often better to use different pointers when accessing different memories.

When converting a version 5.x application to version 6.x, you should first decide which data model, calling convention, and code model that best suit the application. Use Table 1, *Converting from version 5.x memory models* as a guideline. Then find all pointers (generic pointers) in the version 5.x application that are used to access memory not reachable by the version 6.x default pointer. The table below shows which default pointer version 6.x of the compiler uses for the selected data model. Data that cannot be reached with the version 6.x default pointer must be explicitly typed to a pointer type that can access the object in question. Note that if a pointer in the version 5.x application is used to access data in different memories, it might be a good idea to instead use several different pointers. Only when the same pointer needs to access data in different memories should it be explicitly declared as generic.

Data model	Default data memory attribute	Default data pointer attribute
`Tiny`	`__data`	`__idata`
`Small`	`__idata`	`__idata`
`Large`	`__xdata`	`__xdata`
`Generic`	`__xdata`	`__generic`
`Far`	`__far`	`__far`

Table 2: Default pointers used for selected data models in version 6.x

CONSTANTS AND STRINGS

Version 6.x of the compiler offers the same support regarding the location of constants and strings as in version 5.x. However, the default placement and the way the default placement is overridden differ between the two versions. The difference is due to the fact that a generic pointer is no longer always the default pointer in version 6.x, while generic pointers are always the default pointer in version 5.x. In version 5.x constants and strings are by default located in code memory and they are referred to with generic pointers. In version 6.x where the default pointer cannot always reach the code memory, constants and strings can instead be located in the data memory range.

The concept of writable strings (`-y`) in version 5.x is not supported in version 6.x.

In version 6.x of the compiler, the placement of constants and string can be handled in one of three ways:

- Constants and strings are copied from non-volatile memory to volatile memory at system initialization; in this case strings and constants will be handled in the same way as initialized variables. This is the default behavior in version 6.x, and all prebuilt libraries delivered with the product use this method. **Note:** This method requires space for constants and strings both in non-volatile and volatile memory.

- Constants are located in non-volatile memory located in the external data memory addressing range. Thus constants and strings are accessed using the same access method as ordinary external data memory. This is the most efficient method but only possible if the microcontroller has non-volatile memory in the external data memory addressing range. This method is used if the constant or string is explicitly declared as `__xdata_rom/__far_rom/__huge_rom`. This is also the default behavior if the option `--place_constants=data_rom` has been used.

Note: This option has no effect if none of the data models far, generic, or large have been used.

- Constants and strings are located in code memory and are not copied to data memory. This method does not use any additional data memory. However, constants and strings located in code memory can only be accessed through the pointers `__code`, `__far_code`, `__huge_code`, or `__generic`.

When converting an application from version 5.x to version 6.x, select the default placement method that suits the application best. If the microcontroller has non-volatile memory in the extended data memory range, method two should be selected. If the application only uses a small amount of constants and strings and the microcontroller does not have non-volatile memory in the external data memory range, or if it has no external memory at all, method one should be selected. Method three should only be considered if a large amount of strings and constants are used and none of the other two methods are appropriate. There are some complications when using the runtime library together with method three, see *Constants and strings in code memory*, page 15.

The old version 5.x keywords for constant and string location can be converted in the following way:

Version 5.x keyword	Version 6.x keyword
`code`	`__code`
`xdataconst`	`__xdata_rom`

Table 3: Conversion of constant and string location keywords

More information about the placement of constants and strings can be found in the chapters *Compiler options* and *Extended keywords* in the *8051 IAR C/C++ Compiler Reference Guide*.

BYTE ORDER

There are differences in byte order between compiler version 5.x and 6.x that can affect your code. Most applications will not be affected by the change in byte order. However, variables of different types stored at the same memory location, like the `union` construct in C, or access of a part of a variable, may not work correctly, see the example below. Therefore you should review your pointers carefully.

Example

```
#include <stdio.h>
void main(void)
{
  long  l = 0xAABBCCDD;
  char* c;
  int i;

  c = (char*)&l;
  for(i = 0; i < 4; i++)
  {
    printf("0x%X:",*c);
    c++;
  }
}
```

When version 5.x performs memory accesses in big-endian byte order, the program gives the following result:

```
0xAA:0xBB:0xCC:0xDD:
```

When version 6.x adheres to little-endian byte order the result is reversed compared to version 5.x:

```
0xDD:0xCC:0xBB:0xAA:
```

Linker considerations

If you have created your own customized linker command file, we recommend that you create a new one based on a new template supplied in the `\config` directory. Note that many of the segment names have changed, see *Segments*, page 33.

When linking code compiled with version 6.x of the compiler, use the XLINK option `-cx51`. However, when linking code compiled with version 5.x, use the XLINK option `-c8051`.

SEGMENT CONTROL DIRECTIVES

The segment control directive `-b` should not be used together with version 6.x of the compiler. If you were using this directive for locating your banked code in memory, be aware that it is now obsolete and superseded by the new linker segment control directive `-P`. The `-b` directive can be used, but it might not generate the same result as in version 5.x and a warning will be issued.

Syntax

The syntax of the `-P` directive is as follows:

```
P(type)segments=[start-end]*number+increment
```

Parameter	Definition
`type`	Specifies the memory type of the segment
`segments`	A list of banked segments to be linked
`start`	The beginning of the banked segment block
`end`	the end of the banked segment block
`number`	The number of banked segment blocks
`increment`	The increment factor between banks

Example

The following statement declares 4 code banks at the addresses 0x08000–0x0B000, 0x18000–0x1B000, 0x28000–0x2B000, 0x38000–0x3B000.

Version 6.x:

```
--P(CODE)BANKED_CODE=[0x8000-0xB000]*4+10000
```

Version 5.x:

```
-b(CODE)CODE=8000,4000,10000
```

For details of the `-P` directive, see the *IAR Linker and Library Tools Reference Guide*. For more information about placing segments in memory, see the *8051 IAR C/C++ Compiler Reference Guide*.

OBJECT FILE FORMAT

In version 5.x of the compiler, there are a number of command line modifiers used with the command line option `-r` that cause the compiler to include different types of additional information required by the debugger. By default, the version 6.x compiler does not include debugging information, for code efficiency. In version 6.x, when the command line option `-r` or `--debug` is used, source file references are always generated. Embedding of the source code is not supported.

Compiler options

The command line options in version 6.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. For more information, see *Compiler options* in the *8051 IAR C/C++ Compiler Reference Guide*.

Note: A number of new command line options have been added. For a complete list of the available command line options, see *Options summary* in the *8051 IAR C/C++ Compiler Reference Guide*.

MIGRATING PROJECT OPTIONS

Since the available compiler options differ between version 5.x and version 6.x, you should verify your option settings after you have converted an old project.

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

If you are using the IAR Embedded Workbench IDE, you can check the settings in version 5.x and set the corresponding options in version 6.x. Follow these steps:

1. Open the old project in the old IAR Embedded Workbench version.
2. In the project window, select the **Target** you are about to migrate.
3. To save the project settings to a file, right-click in the project window. On the context menu that appears, choose **Save As Text**, and save the settings to an appropriate destination.
4. Use this file and the option tables in this section to verify whether the options you used in your old project are still available or needed. Also check whether you need to use any of the new options.

For information about where to set the equivalent options in the IAR Embedded Workbench IDE, see *Descriptions of options* in the *8051 IAR C/C++ Compiler Reference Guide*.

Removed options

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

Old option	Description
`-C`	Nested comments
`-F`	Form-feed in list file after each function
`-G`	Opens standard input as source; replaced by `-` (dash) as source file name in version 6.x
`-g`	Global strict type checking; in version 6.x, global strict type checking is always enabled
`-gO`	No type information in object code
`-i`	Adds #include file text
`-K`	`//` comments; in version 6.x, `//` comments are allowed unless the option `--strict_ansi` is used
`-P`	Generates PROMable code
`-p`*`nn`*	Lines/page
`-R`	Sets code segment name
`-T`	Active lines only
`-t`*`n`*	Tab spacing
`-U`*`symb`*	Undefines preprocessor symbol
`-X`	Explains C declarations

Table 4: Version 5.x compiler options not available in version 6.x

Identical options

The following table shows the command line options that are *identical* in version 5.x and version 6.x:

Option	Description
`-D`*`symb`*`=`*`value`*	Defines symbols
`-e`	Language extensions
`-f` *`filename`*	Extends the command line
`-I`	Include paths (The syntax is more free in 8051 IAR C/C++ Compiler version 6.x)
`-s[0–9]`	Optimizes for speed
`-z[0–9]`	Optimizes for size

Table 5: Compiler options identical in both compiler versions

Renamed or modified options

The following version 5.x command line options have been *renamed* and/or *modified*:

Version 5.x option	Version 6.x option	Description

`-A`
`-a filename`
	`-la .`
	`-la` *`filename`*
		Assembler output; see *Filenames, page 24.*

`-b`	`--library_module`	Makes an object a library module
`-c`	`--char_is_signed`	`char` is `signed char`

`-E`	`--calling_convention=`
	`idata_reentrant`
		Specifies the default calling convention

`-EL`	`--calling_convention=`
	`xdata_reentrant`
		Specifies the default calling convention

`-ES`	`--calling_convention=`
	`idata_reentrant`
		Specifies the default calling convention

`-gA`	`--strict_ansi`	Flags old-style functions
`-Hname`	`--module_name=`*`name`*	Sets object module name
`-I[`*`prefix`*`]`	`I`*`path`*	Includes paths for `#include` files
`-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

`-mb`	`--code_model=banked,`
	`--data_model=large,`
	`--calling_convention=`
	`xdata_reentrant`
		Uses the banked CODE memory and sets all data to XDATA memory.

`-ml`	`--code_model=near,`
	`--data_model=large,`
	`--calling_convention=`
	`xdata_reentrant`
		Sets the CODE memory to 64 Kbytes of ROM and all data to XDATA memory

`-ms`	`--code_model=near`
	`--data_model=small,`
	`--calling_convention=`
	`idata_overlay`
		Sets the CODE memory to 64 Kbytes of ROM and all data to IDATA

`-mt`	`--code_model=near`
	`--data_model=tiny,`
	`--calling_convention=`
	`data_overlay`
		Sets the CODE memory to 64 Kbytes of ROM and variables to DATA

`-N`*`prefix,`* `-n` *`filename`*	`--preprocess=[c][n][l]`
	`filename`
		Preprocessor output

Table 6: Renamed or modified options

Version 5.x option	Version 6.x option	Description
`-o` *`filename`*, `-O`*`prefix`*	`-o {`*`filename`*`|`*`directory`*`}`	Sets object filename

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

`-r[012][i][n][r][e]`	`-r` `--debug`	Generates debug information; the modifiers have been removed
`-S`	`--silent`	Sets silent operation
`-u`	`--disable_data_alignment`	Disables data alignment of data objects
`-v[0|1]`	`--core={tiny|ti|plain|pl|extended1|e1}`	Specifies the microcontroller core
`-w [s]`	`--no_warnings`	Disables warnings

Table 6: Renamed or modified options (Continued)

Note: There is no exact match between the options `-mc` and `-mm` in version 5.x and the corresponding options in version 6.x. For more information, see Table 1, *Converting from version 5.x memory models*, page 11.

FILENAMES

In version 5.x, file references could 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 6.x, a file reference is always regarded as a *file path* that can be either a directory which the compiler will check and then add a default filename to, or a *filename*.

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

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

Table 7: Specifying filename and directory in version 5.x and version 6.x

LIST FILES

In version 5.x, only one C list file and one assembler list file can be produced; in version 6.x there is no upper limit on the number of list files that can be generated. The new command line option `-l[c|C|a|A][N][H] filename` is used for specifying the behavior of each list file.

ENVIRONMENT VARIABLES

The version 5.x environment variable `QCC8051` has the corresponding environment variable `QCCX51` in version 6.x.

Extended keywords

The set of extended keywords has changed in version 6.x of the compiler. Some keywords have been added, some keywords have been removed, and for some keywords the syntax has changed. In addition, memory attributes have a different interpretation if used in combination with `typedef`.

In version 6.x, all extended keywords start with two underscores, for example `__no_init`.

The following table lists the old keywords, their new equivalents, and completely new keywords:

Keyword in version 5.x	Keyword in version 6.x
`bdata`	`__bdata`
`bit`	`__bit bool`

`code`	`__code`
	`__far_code`
	`__huge_code`

`data`	`__data`
`idata`	`__idata`
`reentrant_idata`	`__idata_reentrant`
`interrupt`	`__interrupt` The interrupt vector *v* and bank register number *n* have to be declared using `#pragma vector=`*v* and `#pragma register_bank=`*n* respectively
`monitor`	`__monitor`
`non_banked`	`__near_func` `__tiny_func`

Table 8: Old and new extended keywords

Keyword in version 5.x	Keyword in version 6.x
`no_init`	`__no_init`
`pdata`	`__pdata`
`plm`	Not available
`sfr`	`__sfr unsigned char`
`xdata`	`__xdata`
`xdataconst`	`__xdata_rom`
`reentrant`	`__xdata_reentrant` `__pdata_reentrant`
`using[`*n*`]`	The bank number *n* has to be declared using `#pragma register_bank=`*n*

Table 8: Old and new extended keywords (Continued)

The incompatibilities between old and new keywords can be solved by using an include file which defines the old names, for instance: `#define near __near`

For detailed information about the extended keywords available in version 6.x, see the chapter *Extended keywords* in the *8051 IAR C/C++ Compiler Reference Guide*.

BITS

To be ANSI-compliant the semantics of bit operations has changed.

In version 5.x a cast from a character to a bit depends on the least significant bit in the byte. For example, assigning a `bool` the value of a `char` with the value `4` (`00000100`) gives the result `0`, because the bit `0` in the `char` is `0`; hence the truncation is applied by the cast. In version 6.x the cast operation acts like a `bool` in the C++ standard for `bool`, which means the cast is interpreted as

```
bit = !(char == 0)
```

The result of the assignment of a `char` variable of value `4` (`00000100`) is `1`, because `!(4 == 0)` gives `1`.

The C++ standard for the `bool` standard also introduces integer promotion.

This means that an expression must give the same result as if all of its operands were of the type `int`. This behavior was not implemented in version 5.x, which means that there might be serious bit problems in your old code.

As an example of the difference between version 5.x and 6.x we examine how the `~bit` construction differs in the two versions.

In version 5.x `~bit` gives `0` if `bit` is `1` and gives `1` if `bit` is `0`. This is not the case in version 6.x.

In version 6.x `~bit` gives `1` if `bit` is `1` and gives `1` if `bit` is `0`. This is because the C++ standard for bool states that the `~bit` operation must act like `bit` where it is of the type `int`. We take a closer look at the example when `bit` is `1`.

The promoted value of `bit` is `1`, which is the same as the binary number `0000000000000001`. The ~ operator toggles the bits and the result is the binary number `1111111111111110` (`0xFFFE`). If—as we explained above—you do the assignment: `bit = ~bit` you get the result `1`, because `!(0xFFFE == 0)` gives `1`.

The ANSI-compliant way to toggle a bit is as follows: `bit = !bit`.

STORAGE MODIFIERS

Both version 5.x and version 6.x of the compiler allow keywords that specify the memory location of an object; the *memory attributes*. Each of these attributes can be used either as a placement attribute for an object, or as a pointer type attribute denoting a pointer that can point to the specified memory.

Replace all storage modifiers used in the version 5.x source code with the corresponding version 6.x keywords, as shown in Table 8, *Old and new extended keywords*. Another declaration syntax is also supported in version 6.x. For example, in version 5.x an `idata` pointer located in xdata memory is declared as:

```
xdata char idata *p;        /* p stored in xdata memory, points
                               to idata memory */
```

In version 6.x the following two declarations correspond to the version 5.x declaration:

```
 __xdata char __idata *p;        /* old syntax */
char  __idata *  __xdata p;      /* recommended syntax */
```

More information about pointer syntax in version 6.x can be found in the *8051 IAR C/C++ Compiler Reference Guide*.

Furthermore, the usage of memory attributes in combination with the keyword `typedef` is stricter in version 6.x than in version 5.x. Version 5.x behaves unexpectedly in some cases:

```
typedef int xdata XINT;
XINT a, b;
XINT data c;        /* Illegal */
XINT *p;            /* p stored in xdata memory, points to
                       default memory type */
```

The first variable declaration works as expected, which means that `a` and `b` are located in xdata memory. However, the declaration of `c` is illegal.

In the last declaration, the `xdata` keyword of the type definition affects the location of the pointer variable `p`, whereas in version 6.x it would have affected the pointer type. Instead the pointer type is default in version 5.x.

The corresponding example for version 6.x is:

```
typedef int __xdata XINT;
XINT a, b;
XINT __data c;        /* c stored in __data memory; override
                         attribute in typedef */
XINT *p;              /* p stored in default memory, points to
                         __xdata memory */
```

The declarations of `c` and `p` differ. The `__data` keyword in the declaration of `c` will always compile. It overrides the keyword of the typedef. In the last declaration the `__xdata` keyword of the typedef affects the type of the pointer. It is thus an `__xdata` pointer to `int`. However, the location of the variable `p` is the default memory location.

INTERRUPT FUNCTIONS AND VECTORS

The syntax for defining interrupt functions has changed from version 5.x.

Old syntax

The syntax when defining interrupt functions using version 5.x:

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

where *`vector`* is the vector offset in the vector table and *`bankno`* is the register bank to be used.

New syntax

The syntax when defining interrupt functions using version 6.x:

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

where *`vector`* is the vector offset in the vector table and *`bankno`* is the register bank to be used.

ABSOLUTE LOCATED VARIABLES

In version 6.x you can locate any object at an absolute address by using the `#pragma location` directive, for example:

```
#pragma location=100
__no_init long PORT;
```

or by using the locator operator `@`, for example:

```
__no_init long PORT @ 100;
```

You can also use the `volatile` attribute on any type, for example:

```
__sfr __no_init volatile unsigned char PORT @ 0x10;
```

In version 5.x the corresponding syntax was:

```
sfr PORT = 0x90;
```

Pragma directives

Version 5.x and version 6.x have different sets of pragma directives for specifying attributes, and they also behave differently:

- In version 5.x, `#pragma memory` specifies the default location of data objects, and `#pragma function` specifies the default location of functions. They change the default attribute to use for declared objects up to the next `#pragma memory` and `#pragma function`; they do not have an effect on pointer types.
- In version 6.x, the `#pragma type_attribute` and `#pragma object_attribute` directives only change the next declared object or `typedef`.

See the chapter *Pragma directives* in the *8051 IAR C/C++ Compiler Reference Guide* for information about the pragma directives available in version 6.x.

Removed pragma directives

The following pragma directives have been removed:

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

These pragma directives are recognized and will give a diagnostic message but will have no effect on the generated code in version 6.x.

Note: Instead of the `#pragma codeseg` directive, we recommend using the `#pragma location` directive or the `@` operator for specifying an absolute location.

Correspondence between old and new pragma directives

The following table shows the mapping of pragma directives from version 5.x to version 6.x:

Pragma directive in version 5.x	Pragma directive in version 6.x
`#pragma bitfields=default`	`#pragma bitfields=default`
`#pragma bitfields=reversed`	`#pragma bitfields=reversed`
`#pragma codeseg (seg_name)`	Not applicable
`#pragma function=default`	Not applicable
`#pragma function=interrupt`	`#pragma type_attribute=__interrupt`
`#pragma function=monitor`	`#pragma type_attribute=__monitor`
`#pragma function=non_banked`	`#pragma type_attribute=__near_func`
`#pragma function=plm`	Not applicable
`#pragma function=reentrant`	`#pragma type_attribute=__xdata_reentrant`
`#pragma function=reentrant_idata`	`#pragma type_attribute=__idata_reentrant`
`#pragma language=default`	`#pragma language=default`
`#pragma language=extended`	`#pragma language=extended`
`#pragma maxargs`	Not applicable
`#pragma memory=code`	`#pragma type_attribute=__code`
`#pragma memory=data`	`#pragma type_attribute=__data`
`#pragma memory=default`	Not applicable
`#pragma memory=idata`	`#pragma type_attribute=__idata`
`#pragma memory=pdata`	`#pragma type_attribute=__pdata`
`#pragma memory=xdata`	`#pragma type_attribute=__xdata`
`#pragma memory=xdataconst`	`#pragma type_attribute=__xdata_rom`
`#pragma memory=constseg(seg_name)`	`#pragma constseg=seg_name`, `#pragma location=`*`address`* (recommended)
`#pragma memory=dataseg(seg_name)`	`#pragma dataseg=seg_name`, `#pragma location=`*`address`* (recommended)
`#pragma memory=no_init`	`#pragma object_attribute=__no_init`
`#pragma overlay=off`	Not applicable
`#pragma overlay=default`	Not applicable
`#pragma stringalloc=xdata`	`#pragma type_attribute=__xdata_rom`

Table 9: Old and new pragma directives

Pragma directive in version 5.x	Pragma directive in version 6.x
`#pragma stringalloc=default`	Not applicable
`#pragma warning=on`	`#pragma diag_warning=`*`tag,tag`*
`#pragma warning=off`	`#pragma diag_suppress=`*`tag,tag`*
`#pragma warning_default`	`#pragma diag_default=`*`tag,tag`*

Table 9: Old and new pragma directives (Continued)

Instead of the pragma directive `#pragma stringalloc=default`, use the compiler option `--place_constants={data|xdata_rom|code}`

Note: The new pragma directives `#pragma type_attribute`, `#pragma location`, `#pragma object_attribute`, `#pragma type_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;
```

New pragma directives

The following pragma directives have been added in version 6.x of the compiler:

- `#pragma constseg`
- `#pragma dataseg`
- `#pragma diag_error`
- `#pragma diag_remark`
- `#pragma diag_suppress`
- `#pragma diag_warning`
- `#pragma data_alignment`
- `#pragma inline`
- `#pragma location`
- `#pragma message`
- `#pragma object_attribute`
- `#pragma optimize`
- `#pragma register_bank`
- `#pragma required`
- `#pragma segment`
- `#pragma type_attribute`
- `#pragma vector`

For information about the new pragma directives, see *Pragma directives* in the *8051 IAR C/C++ Compiler Reference Guide*.

Specific segment placement

In version 5.x of the compiler, the `#pragma memory` directive supports a syntax that enables subsequent data objects that match certain criteria to end up in a specified segment. Each object found after the invocation of a segment placement directive will be placed in this segment, provided that it does not have a memory attribute placement, and that it has the correct constant attribute. For `constseg`, it must be a constant, while for `dataseg`, it cannot be declared `const`.

In version 6.x, the directives `#pragma location` and `#pragma type_attribute`, and the `@` operator are available for this purpose. Note that these attributes affect only the first declaration immediately after the pragma directive.

Intrinsic functions

Version 6.x of the compiler has a new naming convention for intrinsic functions, as well as additional intrinsic functions.

The old intrinsic functions `_args$` and `_argt$` available in version 5.x have been removed and cannot be used in version 6.x.

The following table lists the old intrinsic functions and their new equivalents, as well as the new intrinsic functions:

Intrinsic function in version 5.x	Intrinsic function in version 6.x	Description
`_args$`	None	Returns an array of the parameters to a function
`_argt$`	None	Returns the type of a parameter
None	`__disable_interrupt`	Inserts a disable interrupt instruction
None	`__enable_interrupt`	Inserts an enable interrupt instruction
`_opc`	None	Inserts operation code
None	`__no_operation`	Generates a no operation instruction
None	`__parity`	Indicates the parity of the argument
`_tbac`	`__tbac`	Atomic read, modify, write instruction

Table 10: Old and new intrinsic functions

The `asm` keyword can be used instead of `_opc`, see the *8051 IAR C/C++ Compiler Reference Guide* . See the chapter *Intrinsic functions* in the *8051 IAR C/C++ Compiler Reference Guide* for further information about the intrinsic functions available in version 6.x of the compiler.

Segments

The segment naming convention has changed since version 5.x of the compiler. For information about the segment naming convention used in version 6.x, see *Segments and memory* in the *8051 IAR C/C++ Compiler Reference Guide*.

Version 6.x of the compiler is much more flexible, in the sense that it provides many more options related to variables, constants and calling conventions. Thus many more segments are used in version 6.x; some segments from version 5.x correspond to a certain segment in version 6.x, while other segments from version 5.x can be mapped to one of several segments in version 6.x. Many of the segments in version 6.x are completely new and do not correspond to a segment in version 5.x. For more information about the segments used in version 6.x of the compiler, see *Segment reference* in the *8051 IAR C/C++ Compiler Reference Guide*. The table below shows how segments from version 5.x can be mapped to new segments in version 6.x.

This table lists the old segment names, their counterparts in version 6.x, and additional segments:

Segment in version 5.x	Segment in version 6.x
`BITVARS`	`BIT_N`
`B_CDATA`	`BDATA_ID`
`B_IDATA`	`BDATA_I`
`B_UDATA`	`BDATA_Z`
`C_ARGB`	`DOVERLAY, IOVERLAY, ISTACK`
`C_ARGD`	`DOVERLAY`
`C_ARGI`	`IOVERLAY, ISTACK`
`C_ARGX`	`XSTACK, EXT_STACK`
`CCSTR`[1]	Not available
`C_ICALL`[2]	`BANKED_CODE, NEAR_CODE, TINY_CODE`
`CODE`	`BANKED_CODE, NEAR_CODE, TINY_CODE`
`CONST`	`CODE_AC/C/N`

Table 11: Old and new segments

Segment in version 5.x	Segment in version 6.x
C_RECFN[3]	BANKED_CODE, NEAR_CODE, TINY_CODE
CSTACK	ISTACK
CSTR	BDATA/DATA/IDATA/PDATA/XDATA_I, XDATA_ROM_C, CODE_C
D_CDATA	DATA_ID
D_IDATA	DATA_I
D_UDATA	DATA_Z
ECSTR	Not available
I_CDATA	IDATA_ID
I_IDATA	IDATA_I
I_UDATA	IDATA_Z
INTVEC	INTVEC
NO_INIT	XDATA_AN, XDATA_N
P_CDATA	PDATA_ID
P_IDATA	PDATA_I
P_UDATA	PDATA_Z
RCODE	RCODE
RF_XDATA[4]	XSTACK
X_CDATA	XDATA_ID
X_CONST	XDATA_ROM_AC, XDATA_ROM_C
XCSTR	XDATA_ROM_AC, XDATA_ROM_C
X_IDATA	XDATA_I
X_UDATA	XDATA_Z
XSTACK	XSTACK

Table 11: Old and new segments (Continued)

1) Version 6.x does not support the concept of writable strings. Therefore the segments used in version 5.x for writable strings have no correspondence in version 6.x.
2) In version 6.x functions called indirectly are located in the same segment as ordinary functions. Therefore no special segment is needed for functions called indirectly.
3) In version 6.x recursive functions are located in the same segment as ordinary functions. Therefore no special segment is needed for recursive functions.
4) In version 6.x no special stack is used for recursive functions; the stack specified by the calling convention is used for the specific function. Note that functions using an overlay calling convention cannot be recursive.

ASSEMBLER SOURCE CODE

If you have used any of the segments specific to version 5.x in assembler source code, and if you want to port this assembler code, you must replace all version 5.x segment names with version 6.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 6.x of the 8051 IAR Embedded Workbench IDE.

Other changes

This section describes changes related to:

- Nested comments
- `Sizeof` in preprocessor directives.

NESTED COMMENTS

In version 5.x of the compiler, nested comments are allowed if the option `-C` is used. In version 6.x, nested comments are never allowed. For example, if a comment was used for removing a statement like in the following example, it would not have the desired effect.

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

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

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

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

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

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

Note: `#if` statements may be nested.

SIZEOF IN PREPROCESSOR DIRECTIVES

In version 5.x, `sizeof` could be used in `#if` directives, for example:

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

In version 6.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 *Data types* in the *8051 IAR C/C++ Compiler Reference Guide*.

The size of complex data types may 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);
```