IAR Embedded Workbench

Migration Guide

for Freescale's

HCS12 Microcontroller Family

MHCS12-2

EDITION NOTICE

Second edition: February 2010

Part number: MHCS12-2

This guide applies to version 3.x of IAR Embedded Workbench® for HCS12.

Contents

Tables

Migrating to IAR Embedded Workbench for HCS12 version 3.x

This guide gives hints for porting your application code and projects to the IAR Embedded Workbench IDE for HCS12 version 3.x.

C source code that was originally written for the IAR 68HC12 C Compiler version 2.x can be used also with the new IAR C/C++ Compiler for HCS12 version 3.x. However, some modifications are required. Hereafter, the two compiler versions are referred to as version 2.x and version 3.x, respectively.

This chapter describes the migration considerations and the steps involved.

The migration process

In short, to migrate to version 3.x, consider the following:

- The project file and project setup
- C source code and compiler considerations
- Assembler considerations
- Runtime environment and runtime library
- Linker considerations.

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

Project file and project setup

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

1 Start your new version of IAR Embedded Workbench for HCS12 and create a new workspace by choosing **File>New** and then **Workspace**.

2 Choose **Project>Add Existing Project** to insert your old project into the workspace. This step will create two new project files with the same name as the old file, but with the extensions `ewp` and `ewd`. The `ewp` file contains all settings required to build the application, while the `ewd` file contains all settings related to the debugger. The old project file will remain untouched.

3 It is strongly recommended that you verify that your options have been set up correctly.

To generate a text file with the command line equivalents of the project options in your old project, see *Migrating project options*, page 11.

4 In version 2.x, you can choose between three different memory models:

- The Small memory model; by default, all variables were placed in zpage and all code in non-banked memory
- The Large memory model; by default, all variables were placed anywhere in memory and all code in non-banked memory
- The Banked memory model; by default, all variables were placed anywhere in memory and allowed code size to exceed 64 Kbytes.

In version 3.x, the memory models have been replaced with a new improved mechanism that allows you to control placement of code and data independently.

By default, data can be placed anywhere within the first 64 Kbytes of memory—data16 memory (npage). For individual objects you can use the `__data8` data memory attribute to place the data within the first 256 bytes of memory—data8 memory (zpage).

To control how code is placed in memory you can choose between two different *code models*—Normal for non-banked code, and Banked for banked code. For individual functions it is possible to override the default behavior in each code model by using the function memory attributes `__non_banked` and `__banked`.

Read more about data8 and data16 memory, code models, and memory attributes in the *IAR C/C++ Compiler Reference Guide for HCS12.*

C source code and compiler considerations

In general, version 3.x adheres more strictly to the ISO/ANSI C standard. Most significantly, the checking of data types now adheres more strictly to the ISO/ANSI C standard, compared to version 2.x. This helps you to identify and correct problems in the code, which improves the quality of the object code.

For this reason, it is important to be aware of the fact that code written for version 2.x may generate warnings or errors in version 3.x. A few programming mistakes can generate a vast amount of errors and warnings, where many of them are just consequences of the first ones. Therefore, it is recommended to start with correcting the first one or two errors and then recompile the code. A few measures will most certainly dramatically reduce the number of errors and warnings.

In short, the process of migrating from version 2.x to version 3.x involves the following steps:

1. Replace or modify extended keywords according to the description in the section *Extended keywords*, page 15. To simplify the migration, the file `migration.h` is delivered with version 3.x. This include file maps the old keywords with their new counterparts, if possible.
2. The syntax for specifying interrupt vectors has changed. For more details, see *Interrupt functions and vectors*, page 16.
3. Replace or modify intrinsic functions according to the section *Intrinsic functions*, page 18. To simplify the migration, the file `migration.h` is delivered with version 3.x. Whenever possible, the include file maps old and new intrinsic functions.
4. Replace or modify pragma directives according to the section *Pragma directives*, page 17.
5. Make sure not to use nested comments in your source code. In version 3.x, nested comments are never allowed. For more information, see *Nested comments*, page 4.
6. Version 3.x will by default not accept preprocessor expressions containing any of the following:

   - Floating-point expressions
   - Basic type names and `sizeof`, see *Sizeof in preprocessor directives*, page 5
   - All symbol names (including typedefs, enums and variables).

   With the option `--migration_preprocessor_extensions`, version 3.x will accept such non-standard expressions. For details about this option, see the *IAR C/C++ Compiler Reference Guide for HCS12*.

7 The version 3.x compiler uses a different C parser, and a large number of new optimizations have been added. Depending on your old source code, this might require you to modify your source code. One example of this is a simple delay loop, such as:

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

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

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

8 All predefined symbols supported in version 2.x are also supported in version 3.x. In version 3.x, there are also additional ones. The predefined symbol `__IAR_SYSTEMS_ICC` is provided only for compatibility with version 2.x. Version 3.x also has the `__IAR_SYSTEMS_ICC__` symbol.

See the *IAR C/C++ Compiler Reference Guide for HCS12* for information about the predefined symbols available in version 3.x.

NESTED COMMENTS

In version 2.x, nested comments are allowed if the option `-C` is used. In version 3.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 2.x, `sizeof` can be used in `#if` directives, for example:

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

In version 3.x, `sizeof` is not allowed in `#if` directives (unless the `--migration_preprocessor_extensions` option is used). 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` statement in the source code:

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

To find the size of a predefined data type, see *IAR C/C++ Compiler Reference Guide for HCS12*.

Complex data types may be computed using one of two 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);
  ```

Assembler considerations

In short, consider the following:

1 If your application is written partly in assembler and partly in C, you must consider the calling convention used. For backward compatibility, version 3.x supports the calling convention used by version 2.x—the calling convention *Simple*. Version 3.x also supports a new calling convention—the calling convention *Normal*. To use the calling convention Simple in version 3.x, define and declare your functions with the `__simple` keyword. Both calling conventions are documented in the *IAR C/C++ Compiler Reference Guide for HCS12*.

2 If your application is written partly in assembler and partly in C, and if you have used any of the memory segments specific to version 2.x in assembler source code, you must replace all old segment names with new segment names. For further details, see the section *Segments*, page 19.

3 The assembler environment variables have changed:

Version 2.x assembler	Version 3.x assembler
`ASM_6812`	`ASM_HCS12`
`6812_INC`	`HCS12_INC`

Table 1: Old and new assembler environment variables

4 If your application is written entirely in assembler, you should not include a library in your application. To exclude the library from the build, choose **Project>Options**, select the **General Options** category and click the **Library Configuration** tab. Select **None** from the **Library** drop-down list.

Runtime library and runtime environment

In version 3.x, a new runtime library is provided—the IAR DLIB Library—replacing the runtime library provided with version 2.x—the IAR CLIB Library.

For information about how to migrate from the CLIB library to the DLIB library, see *Migrating from CLIB to DLIB*, page 9. For detailed information about the new library, and the runtime environment it provides, see the *IAR C/C++ Compiler Reference Guide for HCS12*.

To build code produced by version 3.x of the compiler, you must use the runtime environment components it provides.

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 IAR Embedded Workbench for HCS12 version 3.x, this has changed. Now you can configure the runtime library to contain the features that are needed by your application.

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

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

When building an application using 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 (`r12`) there is a corresponding library configuration file (`h`), which has the same name as the library. The files are located in the `hcs12\lib\dlib` directory. The command lines for specifying the library configuration file and library object file could look like this:

```
icchcs12 --dlib_config
<install_dir>\hcs12\lib\dlib\dlhcs12bdn.h

xlink dlhcs12bdn.r12
```

In case you intend to build your own library version, use the default library configuration file `dlhcs12Custom.h`.

To take advantage of all the features, it is recommended that you read about the runtime environment in the *IAR C/C++ Compiler Reference Guide for HCS12*.

PROGRAM ENTRY

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

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

In version 3.x, the default program entry label in `cstartup.s12` 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.s12`.

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 STARTUP AND EXIT CODE—CSTARTUP

In version 3.x, the content of the `cstartup.s12` file has been split up into three files:

`cstartup.s12`, `cmain.s12`, and `cexit.s12`

Now, the `cstartup.s12` file contains the reset vector, initial startup code to setup stacks, and a jump to the `cmain.s12` file. The `cmain.s12` file initializes data segments and executes C++ constructors.

The `cexit.s12` file contains termination code, for example, execution of C++ destructors.

Note: Normally there is no need for customizing the `cmain.s12` file or the `cexit.s12` file.

For old applications that used a modified copy of `cstartup.s33`, you must make a copy of the supplied new `cstartup.s12` file and adapt it to your needs.

MIGRATING FROM CLIB TO DLIB

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

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

You should also see the chapter *The DLIB runtime environment* in the *IAR C/C++ Compiler Reference Guide for HCS12*.

DEVICE-SPECIFIC HEADER FILES

The header files that define peripheral registers delivered with version 2.x can be used with version 3.x. However, for version 3.x applications, it is recommended to use the header files delivered with version 3.x as they are more robust.

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 *Segments*, page 19.

In version 2.x, whenever a module is included in the final link, all functions and variables defined in that module were included. In version 3.x, only those functions and variables that are actually needed, as far as the linker can tell, are included.

If this is a problem, you can use the `__root` attribute in C/C++, or the `ROOT` property in assembler, to force inclusion of particular functions, variables, or segment parts. You can also use the `-g` command line option to XLINK to request inclusion of a certain symbol at link time.

Reference information for migrating to version 3.x

This chapter gives detailed information about the changes in the new version and describes migration considerations related to compiler options, extended keywords, pragma directives, intrinsic functions, and segments.

Compiler options

The command line options in version 3.x follow two different syntax styles:

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

Some options appear in one style only, while other options appear as synonyms in both styles. A number of new command line options have been added. For a complete list of the available command line options, see the *IAR C/C++ Compiler Reference Guide for HCS12*.

The old environment variable `QCC6812` has changed to `QCCHCS12`.

MIGRATING PROJECT OPTIONS

Since the available compiler options differ between version 2.x and version 3.x, you should verify your option settings after you have converted an old project.

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

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

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

1 Open the old project in 6812 IAR Embedded Workbench version 2.x.

2 In the project window, select the project level to get information about options on all levels in your project.

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

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

   For information about where to set the equivalent options in the IAR Embedded Workbench IDE, see the *IAR C/C++ Compiler Reference Guide for HCS12*.

Removed options

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

Old option	Description
`-C`	Nested comments
`-d`	Static locals
`-F`	Form-feed in list file after each function
`-G`	Opens standard input as source; replaced by − (dash) as source file name in version 3.x
`-h`	Disables assignment compatibility attribute test
`-i`	Adds #include file text
`-K`	Enables the use of `'//'` comments; in version 3.x, `'//'` comments are allowed unless the option `--strict_ansi` is used
`-ms`	Small memory model
`-P`	Generates promable code
`-p`*`nn`*	Lines/page
`-T`	Active lines only
`-t`*`n`*	Tab spacing
`-U`*`symb`*	Undefined preprocessor symbol
`-v`	Specifies the microcontroller core
`-X`	Explains C declarations
`-x[DFT2]`	Cross-reference

Table 2: Version 2.x compiler options not available in version 3.x

Identical options

The following table shows the command line options that are *identical* in version 2.x and version 3.x:

Option	Comment
`-Dsymb=value`	Defines symbols
`-e`	Language extensions
`-f filename`	Extends the command line
`-I`	Specifies include paths (The syntax is more free in version 3.x)
`-o filename`	Sets object filename

Table 3: Compiler options identical in both compiler versions

Renamed or modified options

The following version 2.x command line options have been *renamed* and/or *modified*:

Old option	New option	Description

`-A`
`-a filename`
	`-la .`
	`-la filename`
		Assembler output; see *Filenames*, page 14

`-b`	`--library_module`	Makes an object a library module
`-c`	`--char_is_signed`	`'char'` is `'signed char'`
`-gA`	`--strict_ansi`	Flags old-style functions
`-g`, `-gO`	`--omit_types`	No type information in object code
`-Hname`	`--module_name=name`	Sets object module name
`-L[prefix]`, `-l filename`	`-l[a|A|b|B|c|C|D][N][H] {filename|directory}`	Generates list file; the modifiers specify the type of list file to create
`-ml`	`--code_model normal`	Model for non-banked function calls
`-mb`	`--code_model banked`	Model for banked function calls
`-Nprefix, -n filename`	`--preprocess=[c][n][l] filename`	Preprocessor output

`-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
`-R`	`--segment`	Code segment name
`-s[0-9]`	`-s[2|3|6|9]`	Optimizes for speed
`-S`	`--silent`	Sets silent operation

Table 4: Renamed or modified options

Old option	New option	Description
`-w`	`--no_warnings`	Disables warnings
`-z[0-9]`	`-z[2|3|6|9]`	Optimizes for size

Table 4: Renamed or modified options (Continued)

FILENAMES

In version 2.x, file references can be made in either of the following ways:

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

In version 3.x, a file reference is always regarded as a *file path* that can be a directory which the compiler will check and then add a default filename to, or a *filename*.

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

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

Table 5: Specifying filename and directory in version 2.x and version 3.x

LIST FILES

In version 2.x, only one C list file and one assembler list file can be produced; in version 3.x there is no upper limit on the number of list files that can be generated. The new command line option `-l[a|A|b|B|c|C|D][N][H] {`*`filename|directory`*`}` is used for specifying the behavior of each list file.

OBJECT FILE FORMAT

In version 2.x, two types of source references can be generated in the object file. When the command line option `-r` is used, the source statements are being referred to. When the command line option `-re` is used, the actual source code is embedded in the object format.

In version 3.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.

Extended keywords

The set of extended keywords has changed in version 3.x. 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 3.x, all extended keywords start with two underscores, for example `__no_init`.

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

Old keyword	New keyword
`asm`	`asm`, `__asm`
`banked`	`__banked`
`non_banked`	`__non_banked`
`zpage`	`__data8`
`npage`	`__data16`
`interrupt`	`__interrupt`
`monitor`	`__monitor`
`C_task`	`__task`
`no_init`	`__no_init`

Table 6: Old and new extended keywords

To simplify the migration, the include file `migration.h` is delivered with version 3.x. This include file maps the old keywords with their new counterparts, if possible. For example: `#define zpage __data8`

For detailed information about the extended keywords available in version 3.x, see the *IAR C/C++ Compiler Reference Guide for HCS12*.

STORAGE MODIFIERS

Both version 2.x and version 3.x allow keywords that specify the memory location of an object—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.

When the attributes are used directly in the source code, they behave in a similar way in both compiler versions. However, the usage of memory attributes in combination with the keyword `typedef` is more strict in version 3.x than in version 2.x.

Version 2.x behaves unexpectedly in some cases:

```
typedef int zpage MYINT;
MYINT a,b;
MYINT npage c;    /* Illegal */
MYINT *p;         /* p stored in zpage memory, points to default
                     memory type */
```

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

In the last declaration, the `zpage` keyword of the type definition affects the location of the pointer variable `p`, not the pointer type. The pointer type is default.

The corresponding example for version 3.x is:

```
typedef int __data8 MYINT;
MYINT a,b;
MYINT __data16 c; /* c stored in data16 memory; override
                     keyword in type definition */
MYINT *p;         /* p stored in default memory (always data16 in
                     version 3.x), points to data8 memory */
```

The declaration of `c` and `p` differ. The `__data16` keyword in the declaration of `c` will always compile. It overrides the keyword used in the the `typedef` statement. In the last declaration the `__data8` keyword in the `typedef` statement affects the type of the pointer. It is thus a pointer to a data8 `int`. However, the location of the variable `p` is not affected.

INTERRUPT FUNCTIONS AND VECTORS

The syntax for defining interrupt functions has changed from version 2.x. To simplify the conversion from version 2.x syntax to version 3.x syntax, a Perl script is delivered with version 3.x . The script is located in the `hcs12\src\scripts` directory.

Syntax in version 2.x

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

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

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

Syntax in version 3.x

The syntax when defining interrupt functions using version 3.x:

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

where `vector` is an absolute address.

For further details of the new pragma directives, see the *IAR C/C++ Compiler Reference Guide for HCS12*.

Pragma directives

Version 2.x and version 3.x have different sets of pragma directives for specifying attributes, and they also behave differently:

- In version 2.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 until reset back to default behavior or changed to something else; they do not have an effect on pointer types.
- In version 3.x, the `#pragma type_attribute` and `#pragma object_attribute` directives only change the next declared object or `typedef`.

See the *IAR C/C++ Compiler Reference Guide for HCS12* for information about the pragma directives available in version 3.x.

The following pragma directives have been removed:

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

They are recognized and will give a diagnostic message but will not work in version 3.x.

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

The following table shows the mapping of pragma directives:

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

Table 7: Old and new pragma directives

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

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

Specific segment placement

In version 2.x, the #pragma memory directive supports a syntax that enables subsequent data objects that match certain criteria to end up in a specified segment. Each object found after the invocation of a segment placement directive will be placed in the segment, provided that it does not have a memory attribute placement, and that it has the correct constant attribute. For constseg, it must be a constant, while for dataseg, it cannot be declared const.

In version 3.x, the directive #pragma location and the @ operator are available for this purpose.

Intrinsic functions

Version 3.x has a new naming convention for intrinsic functions, as well as a large number of additional functions.

The old intrinsic functions _args$ and _argt$ available in version 2.x are removed and cannot be used in version 3.x. However, except for these two functions, all intrinsic functions available in version 2.x can be used also in version 3.x.

In version 2.x, intrinsic functions are defined in the file in6812.h. To use intrinsic functions in version 3.x, include the file intrinsics.h. To simplify the migration, the file migration.h is delivered with version 3.x. Whenever possible, this include file maps old and new intrinsic functions.

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

Old intrinsic function	New intrinsic function
_args$	--
_argt$	--
_asm	__asm, asm
disable_interrupt	__disable_interrupt
enable_interrupt	__enable_interrupt

Table 8: Old and new intrinsic functions

Old intrinsic function	New intrinsic function
None	`__get_interrupt_state`
None	`__set_interrupt_state`
`wait_for_interrupt`	`__wait_for_interrupt`
`software_interrupt`	`__software_interrupt`
None	`__set_ccr_register`
None	`__get_ccr_register`
`address_24_of`	`__address_24_of`
`min8`	`__min8`
`max8`	`__max8`
`min16`	`__min16`
`max16`	`__max16`
`stop_CPU`	`__stop_CPU`
`_opc`	`__op_code`

Table 8: Old and new intrinsic functions (Continued)

See the *IAR C/C++ Compiler Reference Guide for HCS12* for further information about the intrinsic functions available in version 3.x.

Segments

The segment naming convention has changed since version 2.x. Some of the old segments have been removed, and new ones have been introduced.

For details of the new segments, their names and how they are used, see the *IAR C/C++ Compiler Reference Guide for HCS12*.

This table lists the old segment names, their counterparts in version 3.x, and additional segments:

Old segment	New segment
`CCSTR` [1]	`--`
`CDATA0`	`DATA8_ID`
`CDATA1`	`DATA16_ID`
`CODE`	`CODE`, `BANKED_CODE`
`CONST`	`DATA16_C`
`CSTACK`	`CSTACK`

Table 9: Old and new segments

Old segment	New segment
CSTR	DATA16_C
ECSTR [1]	--
IDATA0	DATA8_I
IDATA1	DATA16_I
NO_INIT	DATA8_N, DATA16_N
RCODE	CODE
UDATA0	DATA8_Z
UDATA1	DATA16_Z
--	DATA8_AC [2]
--	DATA8_AN [2]
--	HEAP
--	DATA16_AC [2]
--	DATA16_AN [2]
--	DIFUNCT
--	INITTAB

Table 9: Old and new segments (Continued)

1) Version 3.x does not support placing strings in writable memory. For this reason, the old segments used for this task have no counterparts in version 3.x.
2) Segments ending in _AN and _AC contain data located at absolute addresses, and should not be included in the linker command file.

Segment control directive -b versus -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*.