IAR Embedded Workbench®

Migrating from UBROF to ELF/DWARF

for the Renesas

RL78 Microcontroller Family

MUBROFELFRL78_I-2

IAR SYSTEMS

EDITION NOTICE
Second edition: April 2016

Part number: MUBROFELFRL78_I-2

Internal reference: ISUD, IJOA.

Contents

Migrating from a UBROF-based product to an ELF/DWARF-based product

This chapter presents the major differences between an IAR Embedded Workbench® product that uses the UBROF object format (hereafter referred to as a UBROF product) and an IAR Embedded Workbench® product that uses the ELF/DWARF object format (hereafter referred to as an ELF/DWARF product), and describes the migration considerations. Primarily, these include how to:

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

For information specific to the migration from version 1.x to version 2.x of IAR Embedded Workbench® for RL78, see *Migrating from v1.x to v2.x of IAR Embedded Workbench for RL78*.

Major difference—the object file format

The main conceptual difference between a UBROF product and an ELF/DWARF product is that the internal object format used by the IAR build tools has changed. In a UBROF product, the IAR Systems format UBROF is used, whereas an ELF/DWARF product uses the industry-standard *Executable and Linkable Format* including DWARF for debug information (ELF/DWARF).

Because the two object formats do not support the same range of features, the tools based on them do not support the same range of features. This is most obvious for the linker. IAR ILINK Linker is dedicated for ELF/DWARF, and it is conceptually very different from the UBROF linker IAR XLINK Linker. There is also a set of tools—referred to as the IAR Utilities—for handling the ELF/DWARF object format files:

- The IAR Archive Tool

- The IAR ELF Tool
- The IAR ELF Dumper
- The IAR Object File Manipulator
- The IAR Absolute Symbol Exporter.

The ILINK linker and the IAR Utilities are described in the *IAR C/C++ Development Guide.*

The benefit of using an ELF/DWARF product is its compatibility (to some extent) with tools from other vendors that also support ELF/DWARF.

The differences in the tools force you to modify your application source code and other related project files. In short, to migrate from a UBROF product to an ELF/DWARF product, you must pay attention to changes in the:

- Compiler and C/C++ source code syntax
- Assembler and assembler source code syntax
- Linker and linker configuration
- Runtime environment and object files
- Project files and project setup in the IDE
- Debugger.

Note that not all issues might be relevant when you migrate your old project. Consider carefully what actions are needed in your case.

Compiler and C/C++ source code syntax

C or C++ source code that was originally written for the IAR C/C++ Compiler in the UBROF product can be used also with the IAR C/C++ Compiler in the ELF/DWARF product. However, some small modifications might be required.

Before compiling existing source code using the new compiler, consider these changes:

- In your C/C++ source code files, this generic syntax change has been made:
  - Depending on your product, the `#pragma vector` directive might no longer be available. To read more about how to specify interrupt vectors, see the *IAR C/C++ Development Guide*.
- Instead of segments, the compiler now places code and data in *sections*. This internal change does not require any changes in your C/C++ source code, unless you are using any of the predefined segment names explicitly in your source code. In that case, you must make sure to use the new section names, see *Segments versus sections*, page 17.

Also, the handling of initialized segments has changed, see *Segments for initialization*, page 17.

- There are some changes related to the compiler options. Some options have been removed, some options have changed, and there are some new options. For a list of changes, see *Tools options*, page 14.
- For information about changes related to filename extensions, see *Filename extensions*, page 20.

Assembler and assembler source code syntax

The name of the assembler executable file has been renamed from `ar178` to `iasmr178`.

In your assembler source code, the following generic changes have been made:

- Modules

  In a UBROF product, you can define one or several assembler modules in each file. In an ELF/DWARF product, there can only be one module per file. This means that you must restructure your files accordingly.

  In the ELF/DWARF product, the assembler cannot make a distinction between program and library *modules*. If you want a module to be treated as a library module, thus conditionally linked, you must place the module in a library.

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

  To read more about modular programming and the new syntax of the module directives, see the *IAR Assembler Reference Guide* supplied with the ELF/DWARF product.

- Segments versus sections

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

  - Assembler directives operating on segments have been either removed or replaced by new directives operating on sections instead, which means you must modify your assembler source code accordingly. For more information, read about section control directives in the *IAR Assembler Reference Guide*
  - If you have used any of the predefined segments specific to the UBROF product in your assembler source code, you must replace all old segment names with new section names. For more information, see *Segments versus sections*, page 17.

- Expressions

  The ELF/DWARF object format restricts the complexity of expressions more than the UBROF object format does. Any affected expressions must be rewritten, otherwise the assembler will generate an error. For more information about expressions, see the *IAR Assembler Reference Guide*.

- Assembler directives

  Some of the assembler directives have been removed and some use a new syntax or have other changes. For a list of assembler directives which are not the same in the ELF/DWARF product as in the UBROF product, see *Assembler directives*, page 19.

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

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

- Predefined symbols

  The predefined symbol `__ASMRL78__` has been replaced by the symbol `__IASMRL78__`.

- Backtrace information for the C-SPY Call stack window

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

- For information about changes related to filename extensions, see *Filename extensions*, page 20.
- The environment variables `ASMRL78` and `ARL78_INC` have changed to `IASMRL78` and `IASMRL78_INC`, respectively.
- There are some changes related to the assembler options. Some options have been removed, some options have changed, and there are some new options. For a list of changes, see *Tools options*, page 14.

For RL78-specific changes, see *Labels in assembler routines*, page 22.

Linker and linker configuration

This section describes the changes in the linker and the linker configuration. For RL78-specific changes, see *Linker considerations when importing old projects*, page 23.

XLINK VERSUS ILINK

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

format. IAR ELF Tool can then transform the executable image that ILINK produced. IAR ELF Tool can add checksums over bytes, fill out areas, or convert the executable image to another format.

In a UBROF product, the compiler places code and data in UBROF *segments*, which XLINK allocates in memory according to directives specified in the *linker configuration file* (filename extension `xcl`). This file is an extension of the command line, which means that you can simply specify any XLINK command line option in it.

In an ELF/DWARF product, the compiler places code and data in ELF *sections*. ILINK allocates these sections according to the *configuration* specified in the *linker configuration file* (filename extension `icf`). This file also supports automatic handling of the application's initialization phase, which means initializing global variable areas and code areas by copying initializers and possibly decompressing them as well. However, the configuration file cannot contain any command line options; these must be specified on the command line.

MIGRATING FROM XLINK TO ILINK

When you migrate from the XLINK linker to the ILINK linker, pay attention to these issues:

- The new IAR ILINK Linker is target-specific which the IAR XLINK Linker is not; the name of the executable file is `xlink` and `ilinkrl78`, respectively.
- To migrate your linker command file to a new ILINK configuration file, see *Converting XLINK.xcl to ILINK.icf*, page 10 for an example.
- For information about how to map segments to sections, see *Segments versus sections*, page 17.
- There are some changes related to the linker options. Some options have been removed, some options have changed, and there are some new options. For a list of changes, see *Tools options*, page 14.
- In a UBROF product you explicitly specify the appropriate system library to use when linking. In an ELF/DWARF product, the appropriate system library is used automatically.
- For information about changes related to filename extensions, see *Filename extensions*, page 20.
- Use IAR ELF Tool if you need to convert the ELF output to either Intel-hex or Motorola S-records.
- In a UBROF product, the `-r` option controls whether debug information should be retained and the `-rt` option controls the level of debug support for C-SPY. In an ELF/DWARF product, all debug information is retained unless the option `--strip` is used and the option `--debug_lib` enables C-SPY debug support.

To learn more about linking, see the *IAR C/C++ Development Guide*.

CONVERTING XLINK.XCL TO ILINK.ICF

Because the linker configuration files for XLINK and ILINK are based on two different paradigms, nothing in the linker configuration file for XLINK is automatically converted. Instead, you must convert your linker setup manually.

We suggest this strategy for converting an XLINK linking setup to an ILINK linking setup:

1. Try to identify the changes in your `xcl` file compared to the original default file.
2. Apply those changes to a default `icf` file using the appropriate mechanisms.

For some general advice, see:

- *Placing sections in memory*, page 11
- *Specifying the size of a section*, page 11
- *Specifying the initialization of sections*, page 11.

In addition, these are some conversion topics you should look at:

- A ROM segment or an uninitialized RAM segment is added in the compiler or assembler. Normally, the default placement is sufficient. If not, add the corresponding section name to the relevant placement command.
- The `-Q` option is used in your XLINK linking setup; replace with the `initialize by copy` linker directive in your ILINK setup.
- A checksum is used; read about the `--checksum` option in the IAR ELF Tool documentation in the *IAR C/C++ Development Guide*.
- Fill is used; read about the `--fill` option in the IAR ELF Tool documentation in the *IAR C/C++ Development Guide*.
- An output format other that ELF is needed; read about IAR ELF Tool in the *IAR C/C++ Development Guide.*
- Overlay is used; read about the `overlay` linker directive in the *IAR C/C++ Development Guide.*
- Far memory placement is used; read about repeating ranges for regions in the *IAR C/C++ Development Guide.*

Placing sections in memory

In the XLINK linker, the placement concept is to place segments, sets of segment parts from one or more modules, into one or more address areas in a memory type (CODE, DATA, etc). For example:

```
-Z(CODE)CODE,CODE2=0-2000
```

The segments are placed in order, in other words the CODE2 segment is placed after the CODE segment. If you use -p, the placement instead becomes unordered.

In the ILINK linker, the placement concept is to place sections, sets of section fragments from one or more modules, by their section type (read/write, read-only, etc) into regions. A region is one or more address areas in a memory. A memory is a specific CPU memory which has a maximum address range. For example:

```
define memory mem with size = 2G;
define region ROM = mem: [from 0x0 to 0xFFFFFF];
place in ROM { ro };
```

The sections have an unordered placement, which means the linker determines the order. To place a section at a specific address you must use the place at linker directive. For example:

```
place at address mem:0x0 { ro section .intvec};
```

Specifying the size of a section

In the XLINK linker, you can specify a constant size to a segment with the + operator. For example:

```
-Z(DATA)CSTACK+8000=8000-A000
```

In the ILINK linker, you must use a block instead of a section to specify a size. For example:

```
define block CSTACK with alignment = 8, size = 0x8000 {};
place in RAM { rw, block CSTACK };
```

With the block directive, it is also possible to specify an ordered set of blocks.

Specifying the initialization of sections

In a UBROF product, a segment that should be initialized during the startup sequence has its initializers placed in a separate segment by the compiler, for example DATA and DATA_I. The placement for both of these segments must be specified in the linker configuration file using either the -Z or the -p option. During the startup sequence, the startup code copies DATA_I to DATA.

In an ELF/DWARF product, the compiler produces only the .data section which also contains the initializers. In the linker configuration file, you must specify how the section should be initialized. Normally, it should be copied in which case you use the initialize by copy linker directive. For example:

```
initialize by copy { rw };
```

For a section that should not be initialized, you must specify the do not initialize directive. For example:

```
do not initialize { section .noinit };
```

In a UBROF product, the startup sequence must explicitly know all segments that somehow should be initialized. In an ELF/DWARF product, the startup sequence uses a table for this and does not know what to initialize. The ILINK linker populates that table with initialization jobs and adds code that executes the initialization jobs.

Project files and project setup in the IDE

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

CONVERTING YOUR PROJECT FILE

The UBROF product and the ELF/DWARF product must be for the same microcontroller.

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

MIGRATING PROJECT OPTIONS

Because the available project options differ between the UBROF product and the ELF/DWARF product, you should verify your option settings after you have converted an old project.

If you are using the command line interface, you can simply compare your makefile with the mapping tables in *Tools options*, page 14, and modify the makefile accordingly.

If you are using the IDE, the options that are the same in both versions might be automatically converted during the project conversion depending on what product and product versions you are using. If automatically converted, the options that have changed will be set to default values.

To verify the options manually, follow this procedure:

1. Choose **Project>Options** to open the **Options** dialog box and select these categories to verify the options:

   - **Compiler** category

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

The **Output** page has changed. If you have defined your own segment name, this will not be automatically converted to a section name. The default code section name is `.mem attr.text`. For more information about segment versus section names, see *Segments versus sections*, page 17.

- **Linker** category

  No linker options are converted automatically. During the project conversion, all linker options will be set to default values. For more information about XLINK options versus ILINK options, see *Differences related to linker options*, page 15. See also *Linker and linker configuration*, page 8.

- **Output Converter** category

  In the UBROF product, XLINK can produce a number of output formats and you specify on the linker **Output** page which one to be used. In the ELF/DWARF product, ILINK produces ELF/DWARF. Use the **Output Converter** options to convert the ELF output to either Intel-hex or Motorola S-records.

- **Library Builder** category

  In the ELF/DWARF product, there is a new library builder, which means no options are converted automatically. During the project conversion, all library builder options will be set to default values.

2 Remember to set any new options.

For information about where to set the equivalent options in the IAR Embedded Workbench IDE, see the online help system.

Runtime environment and object files

INTEROPERABILITY

To build code produced by the compiler in the ELF/DWARF product, you must use the runtime environment components it provides. It is not possible to link object code produced using the ELF/DWARF product with components provided with the UBROF product. This means that you must rebuild your object code from the UBROF product and in some cases you might need to make some source code modifications.

SELECTING RUNTIME LIBRARY FILES

In a UBROF product, you explicitly specify the appropriate system library to use when linking. In an ELF/DWARF product, the appropriate system library is used automatically.

Debugger

FLASH LOADERS

To use a flash loader for downloading your application, an additional output file in the simple-code format is required. In a UBROF product, you must manually set up XLINK to generate this extra `sim` file. In a ELF/DWARF product, this additional file is not required as C-SPY automatically generates the information for the download.

Tools options

This section lists the differences between the command line options in a UBROF product and an ELF/DWARF product, for the compiler, assembler, and the linker.

Note: Only changes due to the changed object format are listed.

DIFFERENCES RELATED TO COMPILER OPTIONS

This table lists the compiler command line options that have changed:

In a UBROF product	Description	In an ELF/DWARF product
`--library_module`	Creates a library module	Removed. To conditionally link a module, it must be part of a library.
`--module_name`	Sets the object module name	Removed
`--omit_types`	Excludes type information	Removed

Table 1: Differences in compiler options

DIFFERENCES RELATED TO ASSEMBLER OPTIONS

This table lists the assembler command line options that have changed (might not concern the product you are using):

In a UBROF product	Description	In an ELF/DWARF product
`-b`	Creates a library module	Removed
`-X`	Unreferenced externals in object files	Removed

Table 2: Differences in assembler options

DIFFERENCES RELATED TO LINKER OPTIONS

This table summarizes the XLINK command line options and their possible counterparts in ILINK:

XLINK option	Description	In ILINK (or IAR ELF Tool)
`-!`	Comment delimiter	In the `icf` file, `/*...*/` or `//`.
`-A`	Loads as program	Removed, see *Assembler and assembler source code syntax*, page 7.
`-a`	Disables static overlay	Removed
`-B`	Always generates output	`--force_output`
`-b`	Defines banked segments. This option was removed in XLINK version 5.1.0.	Removed
`-C`	Loads as a library	Removed, see *Assembler and assembler source code syntax*, page 7.
`-c`	Specifies the processor type	Removed
`-D`	Defines a symbol	`--define_symbol`
`-d`	Disables code generation	Removed
`-E`	Inherent, no object code	Removed
`-e`	Renames external symbols	`--redirect`; note that the syntax has changed.
`-F`	Specifies the output format	Removed
`-f`	Specifies the XCL filename	`-f` extends the command line; `--config` specifies the configuration file.
`-G`	Disables global type checking	Removed
`-g`	Requires global entries	`--keep`
`-H`	Fills unused code memory	`--fill` (IAR ELF Tool option)
`-h`	Fills ranges	`--fill` (IAR ELF Tool option)
`-I`	Specifies the include paths	`--search`
`-J`	Generates a checksum	`--checksum` (IAR ELF Tool option)
`-K`	Duplicates code	Not available
`-L`	Lists to directory	`--log_file`
`-l`	Lists to a named file	`--log_file`
`-M`	Maps logical addresses to physical addresses	Not available

Table 3: Counterparts of XLINK options in ILINK

XLINK option	Description	In ILINK (or IAR ELF Tool)
-n	Ignores local symbols	--no_locals
-O	Multiple output files	Removed
-o	Output file	Unchanged, but --output can also be used as an alias.
-P	Defines packed segments	In the icf file *
-p	Specifies lines/page	Removed
-Q	Scatter loading	In the icf file *
-q	Disables relay function optimization	Removed
-R	Disables range check	--diag_suppress
-r	Debug information	Removed. In ILINK, debug information is included by default, and removed by using --strip.
-rt	Debug information with terminal I/O	--debug_lib
-S	Silent operation	--silent
-s	Specifies a new application entry point	--entry
-U	Address space sharing	Not available
-V	Declares relocation areas for code and data	Removed
-w	Sets diagnostics control	--diag_error, --diag_remark, --diag_suppress, --diag_warning, --diagnostics_tables, --error_list, --no_warnings, --remarks, --warnings_are_errors, --warnings_affect_exit_code
-x	Specifies cross-reference	--map
-Y	Format variant	Removed
-y	Format variant	Removed
-Z	Defines segments	In the icf file *
-z	Segment overlap warnings	Removed

Table 3: Counterparts of XLINK options in ILINK (Continued)

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

Segments versus sections

This section describes the differences between segments in a UBROF product and sections in an ELF/DWARF product.

For a complete list of sections and detailed information about them, see the *IAR C/C++ Development Guide*.

SEGMENTS FOR INITIALIZATION

In a UBROF product, the compiler creates one segment for initializers and one segment for the initialized variables.

In an ELF/DWARF product, the compiler creates one data section that contains the initializers. ILINK then transforms that section for proper handling of the initialization. To read more about initializations, see the *IAR C/C++ Development Guide*.

MAPPING UBROF SEGMENTS TO ELF/DWARF SECTIONS

Some of the UBROF segments have no counterparts in ELF/DWARF, and vice versa.

This table lists the UBROF segments and their counterparts in the ELF/DWARF product:

UBROF segment	ELF/DWARF section	Comments
`CODE`	`.text`	
`RCODE`	`.text`	
`XCODE`	`.textf`	
`INTVEC`	`.intvec`	
`CLTVEC`	`.callt0`	
`HUGE_I`	`.hdata`	
`HUGE_ID`	`.hdata`	Initialization is no longer handled by the compiler. ILINK creates `*_init` sections if needed.
`HUGE_Z`	`.hbss`	
`HUGE_N`	`.hbss.noinit`	
`FAR_I`	`.dataf`	

Table 4: Mapping segments to sections for RL78

UBROF segment	ELF/DWARF section	Comments
FAR_ID	.dataf	Initialization is no longer handled by the compiler. ILINK creates *_init sections if needed.
FAR_Z	.bssf	
FAR_N	.bssf.noinit	
NEAR_I	.data	
NEAR_ID	.data	Initialization is no longer handled by the compiler. ILINK creates *_init sections if needed.
NEAR_Z	.bss	
NEAR_N	.bss.noinit	
SADDR_I	.sdata	
SADDR_ID	.sdata	Initialization is no longer handled by the compiler. ILINK creates *_init sections if needed.
SADDR_Z	.sbss	
SADDR_N	.sbss.noinit	
NEAR_CONST	.const	
FAR_CONST	.constf	
HUGE_CONST	.consth	
SWITCH	.switch	
FSWITCH	.switchf	
WRKSEG	.wrkseg	
OPTBYTE	.option_byte	
SECUID	.security_id	

Table 4: Mapping segments to sections for RL78 (Continued)

Assembler directives

Some of the assembler directives have been removed or have a modified behavior. This table lists the assembler directives which are not the same in an ELF/DWARF product as in a UBROF product:

UBROF product	ELF/DWARF product
ARGFRAME	Removed

Table 5: Differences in assembler directives

UBROF product	ELF/DWARF product
ASEG	Removed
BLOCK	Removed
CFI	The resource names are standardized. A CFI names block must contain a subset of these resource names. See Cfi.m.
COMMON	Removed
DEFFN	Removed
END	No longer takes a program start address as an argument.
ENDMOD	Removed.
FUNCALL	Removed
FUNCTION	Removed
LIBRARY	Removed.
LIMIT	Removed
LOCFRAME	Removed
MODULE	Removed.
NAME	Removed.
ORG	Removed
OVERLOAD	Removed
PROGRAM	Removed.
RSEG	The first instance of the RSEG directive used must not be preceded by any code generating directives, such as DC or DS, or by any assembler instructions. This directive is now an alias for the new directive SECTION. New syntax.
STACK	Removed
SYMBOL	Removed

Table 5: Differences in assembler directives (Continued)

Filename extensions

This table lists the differences related to default filename extensions:

UBROF filename extension	ELF/DWARF filename extension	Description/Comments
s87	s	Assembler source file
r87	o	Object module
r87	a	Library

Table 6: Differences in filename extensions

UBROF filename extension	ELF/DWARF filename extension	Description/Comments
a87	out	Target program
d87	out	Target program for debugging

Table 6: Differences in filename extensions (Continued)

Migrating from v1.x to v2.x of IAR Embedded Workbench for RL78

This chapter presents some of the specific differences between IAR Embedded Workbench® for RL78 version 1.x and IAR Embedded Workbench® for RL78 version 2.x, and describes the primary migration considerations. These are:

- Changes to the calling convention
- Labels in assembler routines
- Linker considerations when importing old projects

Changes to the calling convention

By default, IAR Embedded Workbench® for RL78 version 2.x uses a different calling convention than version 1.x: the V2 calling convention, compliant with the RL78 ABI. To specify this calling convention explicitly, use the function type attribute `__v2_call`.

The legacy V1 calling convention is recommended if you have assembler code written for an earlier version of the compiler. Note that this calling convention has been slightly changed from IAR Embedded Workbench® for RL78 version 1.x: The stack is now cleaned by the calling function, not by the called function, so assembler functions that take stack parameters must be updated before you can use them.

To specify the legacy V1 calling convention for an individual function, use the function type attribute `__v1_call`.

If an assembler routine will be used by both versions of IAR Embedded Workbench® for RL78, you can use the predefined preprocessor symbol `__VER__` to identify which version that is used, and how the stack is cleaned up:

```
#if __VER__ >= 200
    ; Caller is responsible for stack clean up of parameters
#else
    ; Callee is responsible for stack clean up of parameters
#endif
```

Note: Remember to use the function type attribute `__v1_call` when you port any v1.x C prototypes for assembler functions.

Migrating assembler code

Before you start, read through the assembler-related information in *Migrating from a UBROF-based product to an ELF/DWARF-based product*, page 5. Many directives have been removed and some expressions are too complex to be represented in ELF.

MIGRATING ASSEMBLER MODULES

In IAR Embedded Workbench® for RL78 version 2.x there can only be one assembler module per source file. The linker will remove any unreferenced `SECTION` part just like it removed unreferenced modules.

These major changes apply to the assembler in IAR Embedded Workbench® for RL78 version 2.x:

- The assembler directives `MODULE`, `PROGRAM`, `NAME`, `LIBRARY`, and `ENDMOD` are no longer supported and must be removed. Remember to end the file with `END`.
- The names of local labels in the various assembler modules must be unique within a section, because they will all exist in the same file context.
- `!` or `!!` can no longer be used for getting absolute addresses. Use `N:` instead as a prefix for 16-bit addresses, and `F:` as a prefix for 20-bit addresses.
- Include files must always be included using the `#include` directive. The `$INCLUDE` symbol can no longer be used.
- The syntax of the `RSEG` and `SECTION` directives now requires a memory type after the section name, for example: `section TABLE:CONST`
- `NOT` (or `!`) in assembler instructions is only supported if the expression can be calculated during the assembly.

LABELS IN ASSEMBLER ROUTINES

The change from the XLINK Linker in version 1.x to the ILINK Linker in version 2.x has some consequences for the use of labels in assembler routines.

The ILINK Linker performs a range check of the values of labels, which means that labels placed in near memory (`0xF0000–0xFFFFF`) will produce a linker error unless you have handled them correctly in the code.

For example:

```
MOVW HL,#my_near_label
```

will produce a linker error because the value of `my_near_label` is too large for the 16-bit register `HL`.

The correct way to write this is:

```
MOVW HL,#LWRD(my_near_label)
```

which is also backward compatible with IAR Embedded Workbench® for RL78 version 1.x.

To support the RL78 ABI (the Application Binary Interface for RL78), the compiler in IAR Embedded Workbench® for RL78 version 2.x generates assembler labels for symbol and function names by prefixing an underscore. Thus, if an assembler routine is called from C, you must prefix the label with _. For example:

C prototype:

```
void test();
```

Assembler declaration:

```
public _test
_test:
   ret
```

Linker considerations when importing old projects

The change from the generic XLINK Linker in version 1.x to the RL78-specific ILINK Linker in version 2.x means that the linker settings will be reset.

In particular, you must check the settings for:

- Buffered write
- Map file generation
- Checksum configuration
- Diagnostic messages
- RAW binary images
- Defined symbols. Note that the ILINK Linker has two kinds of defines: one for use by the linker configuration file, and one for labels referred to from your code.

For more information about migrating to the ILINK Linker, see *Linker and linker configuration*, page 8, *Differences related to linker options*, page 15, and *Segments versus sections*, page 17.

Linking object files produced by Renesas

The IAR ILINK Linker places sections based on their names. Object files produced by Renesas contain both a section name and a section type. ILINK generates additional section names based on the section type, so that Renesas object files are linked to the correct memory even if the section has a custom name.

To override this behavior, specify `section` *`MYNAME`* (where *`MYNAME`* is the custom section name) at the preferred memory location in the linker configuration file. The generated names are all of the type `R_`*`type`* where *`type`* is one of:

```
CALLT0
TEXT
TEXTF
TEXTF_UNIT64KP
CONST_init
CONST
CONSTF
SDATA
SBSS
DATA
BSS
DATAF
BSSF
AT
DATA_AT
BSS_AT
OPT_BYTE
SECUR_ID
```