ARM® RealView Developer Suite

to

ARM IAR Embedded Workbench®

Migration Guide

Part no: EWM-RVARM-1b

Introduction

This guide examines the differences between using the RealView ARM® development tools and the IAR Systems ARM development tools. The issues related to assembler conversion range from basic topics such as command line options, system segment/area names, listing/output options, code generation options, register naming differences, assembler operators, assembler directives, pseudo-instructions, and other assembler differences, to advanced topics such as predefined symbols, conditional assembly, macros, and modules. Linker related topics such as command line options and image memory mapping are also documented.

The features, options, descriptions, and examples specified in the document are based on tools associated with *ARM RealView Developer Suite Version 2.2* and *ARM IAR Embedded Workbench Version 4.20A.*

Information about the RealView development tools was obtained from the *RealView Compilation Tools Version 2.2 Assembler Guide* (ARM DUI 0204E) and the *RealView Compilation Tools Version 2.2 Linker and Utilities Guide* (ARM DUI 0206E). Information about the IAR Systems development tools is based on the *ARM IAR Assembler Reference Guide* (AARM-6) and the *IAR Linker and Library Tools Reference Guide* (XLINK-459I).

IAR Embedded Workbench IDE overview

The IAR Embedded Workbench IDE consists of tools such as a compiler, assembler, linker, library builder, librarian, editor, project manager, command line interface, debugger, and simulator.

RealView Developer Suite includes a compilation tool set (RealView Compilation Tools (RVCT), which includes a compiler, assembler, linker, librarian, editor, project manager and command line interface), debugger and an Instruction Set Simulator (ISS).

Equivalent tools from both development environments are listed in table 1 together with the command line commands for invoking them:

Tools	RealView Developer Suite	IAR Embedded Workbench IDE
Compiler	RVCT C/C++ compiler, `armcc`	IAR C/C++ Compiler, `iccarm`
Assembler	RVCT ARM assembler, `armasm`	IAR ARM Assembler, `aarm`
Linker	RVCT ARM linker, `armlink`	IAR XLINK Linker, `xlink`
Library builder	-	IAR XAR Library Builder, `xar`
Librarian	RVCT ARM librarian, `armar`	IAR XLIB Librarian, `xlib`
Debugger	RV Debugger	IAR C-SPY® Debugger
Simulator	RV ARMulator ISS	IAR C-SPY® Simulator

1. RealView Developer Suite and IAR Embedded Workbench IDE equivalents

The ARM IAR C/C++ Compiler features efficient code generation with debug information, C/C++ language support facilities and type checking.

The ARM IAR Assembler features a built-in C preprocessor and supports conditional assembly.

The IAR XLINK Linker links object files produced by the compiler or assembler to produce machine code for the ARM core, while the IAR XAR Library Builder and IAR XLIB Librarian allow manipulation of library object files.

The IAR C-SPY® Debugger is a high-level language debugger that is integrated into the IDE, so corrections are made directly in the same source code window used to control the debugging.

TOOLS COMPARISON

The primary difference between the tool sets is the level of integration of the debugger with the rest of the IDE. In IAR Embedded Workbench, the C-SPY Debugger is completely integrated with the IDE, whereas in RealView Developer Suite, the RV Debugger is more standalone. However, essential project management options and tools, and make/build/debug menus are similar.

General debugger features like source and disassembly level debugging, source stepping, setting breakpoints, variable, register and expression monitoring/watching, call stack information and facilities for third-party extensions (RTOS awareness, simulation modules, emulator drivers, etc.) are available in both tool sets. Both debugger text editors provide typical utilities such as colored keywords and search and replace.

For Microsoft® Windows®, a features-equivalent RealView Developer Suite editor is Metrowerks™ Codewarrior™. Project files created with Codewarrior have the filename extension `.mcp`, compared to RealView Developer Suite project files that have the extension `.prj`. Note that if you migrate from RealView Developer Suite to Codewarrior tools or vice versa, the project file will need to be re-created and relevant source files re-added.

Getting started

This section discusses how to get started with converting C and assembler projects from RealView Developer Suite to IAR Embedded Workbench.

Filename extensions

In RealView Developer Suite, projects list all associated source files required by the target application. These project files have a `.prj` filename extension. In IAR Embedded Workbench, workspaces are used for organizing multiple projects. This is useful when you are simultaneously managing several related projects. Workspace files have the filename extension `.eww`, and project files have the extension `.ewp` in IAR Embedded Workbench.

The filename extensions of C source and header files are `.c` and `.h`, respectively, in both RealView Developer Suite and IAR Embedded Workbench, which includes standard library files and user-specific files.

The filename extension of assembler source files in RealView Developer Suite is `.s`. IAR Embedded Workbench uses `.s79` by default, but in addition accepts the `.s` extension.

The object files produced by the compiler or assembler have the filename extension `.o` (in RealView Developer Suite) or `.r79` (in IAR Embedded Workbench.

Converting assembler source files

The guidelines in the following sections describe how to accurately and systematically convert assembler source files from RealView Developer Suite to IAR Embedded Workbench.

BASIC ASSEMBLER CONVERSION

For basic assembler conversion, use the following steps, shown with a simple example:

1) Redefine system segments and areas.

Before step 1:

```
                AREA test,CODE
                .
index           RN 9
                LDR r0, [index,#4]
                .
                LDR r1,=&FF00
                .
                AREA hash,DATA
                .
                DCFD 12.3
                .
value           EQU 8
                LDR r5,=value || 0xF9
                .
                END
```

After step 1:

```
                RSEG test:CODE:NOROOT(2)
                .
index           RN 9
                LDR r0, [index,#4]
                .
                LDR r1,=&FF00
                .
                RSEG hash:DATA:NOROOT(2)
                .
                DCFD 12.3
                .
value           EQU 8
                LDR r5,=value || 0xF9
                .
                END
```

2) Remove use of the RealView Developer Suite `RN` directive. Rename registers (if required).

Before step 2:

```
                RSEG test:CODE:NOROOT(2)
                .
index           RN 9
                LDR r0, [index,#4]
                .
                LDR r1,=&FF00
                .
                RSEG hash:DATA:NOROOT(2)
                .
                DCFD 12.3
                .
value           EQU 8
                LDR r5,=value || 0xF9
                .
                END
```

After step 2:

```
                RSEG test:CODE:NOROOT(2)
                .
                LDR r0, [r9,#4]
                .
                LDR r1,=&FF00
                .
                RSEG hash:DATA:NOROOT(2)
                .
                DCFD 12.3
                .
value           EQU 8
                LDR r5,=value || 0xF9
                .
                END
```

3) Modify unary and binary assembler operators, while noting operator precedence. The example shows the modification of the bitwise `OR` operator from `||` (in RealView Developer Suite) to `|` (in IAR Embedded Workbench).

Before step 3:

```
                RSEG test:CODE:NOROOT(2)
                .
                LDR r0, [r9,#4]
                .
                LDR r1,=&FF00
                .
                RSEG hash:DATA:NOROOT(2)
                .
                DCFD 12.3
                .
value           EQU 8
                LDR r5,=value || 0xF9
                .
                END
```

After step 3:

```
                RSEG test:CODE:NOROOT(2)
                .
                LDR r0, [r9,#4]
                .
                LDR r1,=&FF00
                .
                RSEG hash:DATA:NOROOT(2)
                .
                DCFD 12.3
                .
value           EQU 8
                LDR r5,=value | 0xF9
                .
                END
```

4) Modify assembler directives

Before step 4:

```
                RSEG test:CODE:NOROOT(2)
                .
                LDR r0, [r9,#4]
                .
                LDR r1,=&FF00
                .
                RSEG hash:DATA:NOROOT(2)
                .
                DCFD 12.3
                .
value           EQU 8
                LDR r5,=value :OR: 0xF9
                .
                END
```

After step 4:

```
                RSEG test:CODE:NOROOT(2)
                .
                LDR r0, [r9,#4]
                .
                LDR r1,=&FF00
                .
                RSEG hash:DATA:NOROOT(2)
                .
                DF64 12.3
                .
value           EQU 8
                LDR r5,=value :OR: 0xF9
                .
                END
```

5) Modify assembler symbols, numeric literals and numeric expressions (if required). Note that assembler pseudo-instructions and labels do not need to be modified. The example below shows the modification of a numeric literal.

Before step 5:

```
                RSEG test:CODE:NOROOT(2)
                .
                LDR r0, [r9,#4]
                .
                LDR r1,=&FF00
                .
                RSEG hash:DATA:NOROOT(2)
                .
                DF64 12.3
                .
value           EQU 8
                LDR r5,=value :OR: 0xF9
                .
                END
```

After step 5:

```
                RSEG test:CODE:NOROOT(2)
                .
                LDR r0, [r9,#4]
                .
                LDR r1,=0xFF00
                .
                RSEG hash:DATA:NOROOT(2)
                .
                DF64 12.3
                .
value           EQU 8
                LDR r5,=value :OR: 0xF9
                .
                END
```

COMPLEX ASSEMBLER CONVERSION

For more complex assembler conversions, follow the steps outlined below. (Detailed descriptions and associated examples have been provided in the section *Advanced conversion* on page 16.)

1) Modify predefined symbols.
2) Modify conditional assembly directives.
3) Convert macros.
4) Create modules (if required).

Makefiles

The following steps describe the method of converting makefiles from RealView Developer Suite to IAR Embedded Workbench. A simple example of a makefile conversion is provided.

1) Change the assembler to use from `armasm` (RealView Developer Suite) to `aarm` (IAR Embedded Workbench)

Before step 1:

```
#Assembler to use
AS=armasm
#Options to pass to the assembler
AFLAGS=-g --bigend --list=test

hello.o: hello.s
    $(AS) $(AFLAGS) hello.s

clean:
    rm -rf *.o
```

After step 1:

```
#Assembler to use
AS=aarm
#Options to pass to the assembler
AFLAGS=-g --bigend --list=test

hello.o: hello.s
    $(AS) $(AFLAGS) hello.s

clean:
    rm -rf *.o
```

2) Modify command line options. The example shows how to change the `-g` option used for generating debug information in RealView Developer Suite to the equivalent option in IAR Embedded Workbench, `-r`.

Before step 2:

```
#Assembler to use
AS=aarm
#Options to pass to the assembler
AFLAGS=-g --bigend --list=test

hello.o: hello.s
    $(AS) $(AFLAGS) hello.s

clean:
    rm -rf *.o
```

After step 2:

```
#Assembler to use
AS=aarm
#Options to pass to the assembler
AFLAGS=-r --bigend --list=test

hello.o: hello.s
    $(AS) $(AFLAGS) hello.s

clean:
    rm -rf *.o
```

3) Modify code generation options. The example shows how to change the RealView Developer Suite option for generating big-endian ordered code and data to the equivalent option in IAR Embedded Workbench.

Before step 3:

```
#Assembler to use
AS=aarm
#Options to pass to the assembler
AFLAGS=-r --bigend --list=test

hello.o: hello.s
    $(AS) $(AFLAGS) hello.s

clean:
    rm -rf *.o
```

After step 3:

```
#Assembler to use
AS=aarm
#Options to pass to the assembler
AFLAGS=-r --endian big --list=test

hello.o: hello.s
    $(AS) $(AFLAGS) hello.s

clean:
    rm -rf *.o
```

4) Modify listing/output options. The example shows how to change the RealView Developer Suite option for producing a listing output file to the equivalent option in IAR Embedded Workbench.

Before step 4:

```
#Assembler to use
AS=aarm
#Options to pass to the assembler
AFLAGS=-r --endian big --list=test

hello.o: hello.s
    $(AS) $(AFLAGS) hello.s

clean:
    rm -rf *.o
```

After step 4:

```
#Assembler to use
AS=aarm
#Options to pass to the assembler
AFLAGS=-r --endian big -l test

hello.o: hello.s
    $(AS) $(AFLAGS) hello.s

clean:
    rm -rf *.o
```

Linker Files

Converting linker files from RealView Developer Suite to IAR Embedded Workbench is similar to the conversion of makefiles. Refer to the section *Linker and other tools* on page 19 for a detailed description of linker options and the memory mapping mechanism.

Follow these steps:

1) Modify linker command line options.
2) Change the memory mapping method from using scatter loading (RealView Developer Suite) to segment control (IAR Embedded Workbench).

Migration reference

This section lists the differences in assembler, compiler, and linker options between ARM® RealView Developer Suite and ARM IAR Embedded Workbench®.

Assembler conversion

In ARM RealView Developer Suite, the assembler is called `armasm`, while in ARM IAR Embedded Workbench, the assembler is called `aarm`.

COMMON ASSEMBLER COMMAND LINE OPTIONS

The following table lists the commonly used command line options.

RealView	IAR	Description
`--apcs [`*`qualifiers`*`]`	No equivalent	Specifies which *Procedure Call Standard for the ARM Architecture* (AAPCS) that is being used
`--bigend or --bi`	`-e`	Generates code in big-endian byte order
`--bigend or --bi, --littleend or --li`	`--endian{little|l|big|b}`	Specifies the byte order of the generated code and data
`--cpu` *`name`*	`--cpu` *`name`*	Specifies the target CPU or core
`-i` *`dir`* `[,`*`dir`*`]...`	`-I`*`prefix`*	Adds directories to the include file search path
`-g`	`-r[en]`	Instructs the assembler to generate debug information; `-re` includes the full source file into the object file and `-rn` generates an object file without source information
`--list [`*`filename`*`]`	`-l` *`filename`*	Instructs the assembler to generate a listing
`-m`	No equivalent	Instructs the assembler to write source file dependency lists to `stdout`
`-o` *`filename`*	`-o` *`filename`*	Sets the output object filename
`--via` *`file`*	`-f extend.xcl`	Instructs the assembler to open and read command line arguments from a file
`--xref or -x`	`-x{DI2}`	Instructs the assembler to list cross-reference information; In ARM IAR Embedded Workbench, `-xD` includes `#define` references, `-xI` includes internal symbols and `-x2` includes dual line spacing

2. Common command line options in RealView Developer Suite and ARM IAR Embedded Workbench

DEFINING SYSTEM SEGMENTS/AREAS

System segments and areas are defined with the `AREA` directive in RealView Developer Suite. In ARM IAR Embedded Workbench, the equivalent directive is called `RSEG`.

In ARM IAR Embedded Workbench, `ORG` is used to set the program location counter of the current segment to the value of an expression. There is no support for `ORG` in RealView Developer Suite. Instead, RealView Developer Suite uses either the `armlink` option `--first` or scatter loading.

The example below compares the methods of defining system segments/areas in RealView Developer Suite and ARM IAR Embedded Workbench.

RealView	IAR	Description
`AREA` *`test`*`,CODE`	`RSEG` *`test`*`:CODE:NOROOT(2)`	`;Assembles a new code section called` *`test`*
`.`	`.`	
`MOV R0,#10`	`MOV R0, #10`	`;Set up a parameter`
`LDR R3,=0x1234`	`LDR R3,=0x1234`	`;Load  0x1234 into register R3`
`.`	`.`	
`.`	`.`	
`END`	`END`	`;End of source file`

3. Defining system segments/areas in RealView Developer Suite and ARM IAR Embedded Workbench

LISTING/OUTPUT OPTIONS

In both RealView Developer Suite and ARM IAR Embedded Workbench, the `-o` command line option sets the filename to be used for the output object file. If no filename argument (or extension) is defined, the assembler creates an object filename of the form *`inputfilename`*`.o` (in RealView Developer Suite) or *`inputfilename`*`.r79` (in ARM IAR Embedded Workbench).

In order to instruct the assembler to generate a detailed list file of the assembler code it produced, the `--list` option in RealView Developer Suite or the `-l` option in ARM IAR Embedded Workbench can be used. By default, the assembler does not generate a list file.

The behavior of `--list` (in RealView Developer Suite) and `-l` (in IAR) can be controlled with the cross-reference option. In RealView Developer Suite, the `--xref` (or `-x`) command line option instructs the assembler to list cross-reference information about where symbols were defined and where they were used, both inside and outside macros. In comparison, the `-x` option in IAR Embedded Workbench makes the assembler include a cross-reference table at the end of the list file. Additionally, IAR Embedded

Workbench provides the following parameters: `-xD` for inclusion of `#define` symbols, `-xI` for inclusion of internal symbols and `-x2` for inclusion of dual line spacing.

CODE GENERATION OPTIONS

In RealView Developer Suite, the `--apcs` command line option can be used to specify the attributes of code sections. There is no equivalent command line option in IAR Embedded Workbench. Valid qualifiers for `--apcs` are provided in the table below.

[qualifier]	Description
`/none`	Input file does not use AAPCS
`/interwork` or `/inter`	Code in the input file is suitable for ARM/Thumb interworking
`/nointerwork` or `/nointer`	Code in the input file is not suitable for ARM/Thumb interworking
`/ropi` or `/pic`	Content of the input file is read-only position-independent
`/noropi` or `/nopic`	Content of the input file is not read-only position-independent (default)
`/rwpi` or `/pid`	Content of the input file is read-write position-independent
`/norwpi` or `/nopid`	Content of the input file is not read-write position-independent (default)
`/swstackcheck` or `/swst`	Code in the input file performs software stack-limit checking
`/noswstackcheck` or `/noswst`	Code in the input file does not perform software stack-limit checking (default)
`/swstna`	Code in the input file is compatible with code that performs and does not perform software stack-limit checking

4. Qualifiers for the --apcs command line option in RealView Developer Suite

The `--bigend` (or `--bi`) and `--littleend` (or `--li`) options in RealView Developer Suite specify the byte order of the generated code or data, while the equivalent option in IAR Embedded Workbench is `--endian{little|l|big|b}`. Furthermore, the `-e` option in IAR Embedded Workbench can also be used to generate code in big-endian byte order. The default byte order in both ARM RealView Developer Suite and ARM IAR Embedded Workbench is little-endian.

The `--cpu` command line option is used in both RealView Developer Suite and IAR Embedded Workbench to specify the target core and obtain the correct instruction set. The default CPU name is `ARM7TDMI` in both RealView Developer Suite and IAR Embedded Workbench.

REGISTER NAMING DIFFERENCES

The following table lists the register naming differences between RealView Developer Suite and IAR Embedded Workbench. Note that the assembler option `-j` (for allowing alternative register names, mnemonics, and operands) is needed to allow the use of the register names `A1`–`A4`, `V1`–`V8`, `SB`, `SL`, `FP`, and `IP` in IAR Embedded Workbench.

RealView	IAR	Description
`r0`, `R0`, and `a1`	`R0` and `A1`	Argument, result or scratch register
`r1`, `R1`, and `a2`	`R1` and `A2`	Argument, result or scratch register
`r2`, `R2`, and `a3`	`R2` and `A3`	Argument, result or scratch register
`r3`, `R3`, and `a4`	`R3` and `A4`	Argument, result or scratch register
`r4`, `R4`, and `v1`	`R4` and `V1`	Variable register
`r5`, `R5`, and `v2`	`R5` and `V2`	Variable register
`r6`, `R6`, and `v3`	`R6` and `V3`	Variable register
`r7`, `R7`, and `v4`	`R7` and `V4`	Variable register
`r8`, `R8`, and `v5`	`R8` and `V5`	Variable register
`r9`, `R9`, and `v6`	`R9` and `V6`	Variable register
`r10`, `R10`, and `v7`	`R10` and `V7`	Variable register
`r11`, `R11`, and `v8`	`R11`	Variable register
`r12` and `R12`	`R12`	General purpose register
`sb` and `SB`	`SB`	Static base, `r9`
`sl` and `SL`	`SL`	Stack limit, `r10`
`fp` and `FP`	`FP`	Frame pointer, `r11`
`ip` and `IP`	`IP`	Intra-procedure-call scratch register, `r12`
`sp` and `SP`	`R13` (`SP`)	Stack pointer, `r13`
`lr` and `LR`	`R14` (`LR`)	Link register, `r14`
`pc` and `PC`	`R15` (`PC`)	Program Counter, `r15`
`cpsr` and `CPSR`	`CPSR`	Current program status register
`spsr` and `SPSR`	`SPSR`	Saved progress status register

5. Register naming differences in RealView Developer Suite and IAR Embedded Workbench

ASSEMBLER OPERATORS

RealView Developer Suite and IAR Embedded Workbench possess many operators in common, and shift and mask operators can be used to implement many of the missing operators.

Operator precedence

The assemblers in RealView Developer Suite and IAR Embedded Workbench use extensive sets of operators. Operators with the highest precedence are evaluated first, followed by the operators with the second highest precedence and so forth until the lowest precedence operators are evaluated. If an expression contains operators of equal precedence, the operators are evaluated from left to right. In RealView Developer Suite and IAR Embedded Workbench both, the parentheses (and) can be used for grouping operators and operands and to denote precedence.

The table below shows the order of precedence (from top to bottom) of operators in both development environments.

RealView	IAR
Unary operators	Unary operators
Multiplicative arithmetic operators	Multiplicative arithmetic operators
String manipulation operators	Addition and subtraction operators
Shift operators	Shift operators
Addition, subtraction and logical operators	Logical AND operators
Relational operators	Logical OR operators
Boolean operators	Relational operators

6. Operator precedence in RealView Developer Suite and IAR Embedded Workbench

Unary operators

The following table shows the equivalent assembler unary operators in RealView Developer Suite and IAR Embedded Workbench. Note that IAR Embedded Workbench does not have any unary operators that return strings, only numeric or logical values.

RealView	IAR	Description
Returns strings		
`:CHR:`	No equivalent	ASCII character return
`:LOWERCASE:`	No equivalent	Converts all uppercase characters to lowercase
`:REVERSE_CC:`	No equivalent	Inverse of condition code
`:STR:`	No equivalent	Numeric expression: Returns 8-digit hex string Logical expression: Returns `"T"` or `"F"`
`:UPPERCASE:`	No equivalent	Converts all lowercase characters to uppercase
Returns numeric or logical values		
+	+	Unary plus
-	-	Unary minus
`:LNOT:`	`!` or `:LNOT:`	Logical complement
`:NOT:`	~ or `:NOT:`	Bitwise complement
No equivalent	`LOW`	Low byte
No equivalent	`HIGH`	High byte
No equivalent	`BYTE1`	First byte
No equivalent	`BYTE2`	Second byte
No equivalent	`BYTE3`	Third byte
No equivalent	`BYTE4`	Fourth byte
No equivalent	`LWRD`	Low word
No equivalent	`HWRD`	High word
No equivalent	`DATE`	Current time/date
No equivalent	`SFB`	Segment begin
No equivalent	`SFE`	Segment end
No equivalent	`SIZEOF`	Segment size
?	No equivalent	Number of bytes of executable code generated by line defining a symbol, for example `?A`
`:BASE:`	No equivalent	Number of register component
`:CC_ENCODING:`	No equivalent	Numeric value of condition code
`:DEF:`	No equivalent	If defined, `TRUE`, else `FALSE`
`:INDEX:`	No equivalent	Offset from base register
`:LEN:`	No equivalent	Length of string
`:RCONST:`	No equivalent	Number of register, `0–15` (i.e. `r0–r15`)
`:SB_OFFSET_19_12:`	`(:SHR:) :AND: 0xFF`	Bits `[19:12]`
`:SB_OFFSET_11_0:`	`:AND: 0xFFF`	Least significant 12 bytes

7. Unary operators in RealView Developer Suite and IAR Embedded Workbench

Binary operators

The following table shows the equivalent assembler binary operators in RealView Developer Suite and IAR Embedded Workbench.

RealView	IAR	Description
Multiplicative Arithmetic Operators		
*	*	Multiplication
/	/	Division
% or :MOD:	% or :MOD:	Modulo
String Manipulation Operators		
:CC:	No equivalent	Concatenate
:LEFT:	No equivalent	Left-most characters
:RIGHT:	No equivalent	Right-most characters
Shift Operators		
:ROL:	No equivalent	Logical rotation left. In IAR Embedded Workbench, there is no direct equivalent, but can be achieved with the following (x :SHL: 1) :OR: (x :SHR (32-1))
:ROR:	No equivalent	Logical rotation right. In IAR Embedded Workbench, there is no direct equivalent, but can be achieved with the following (x :SHR: 1) :OR: (x :SHL (32-1))
<< or :SHL:	<< or :SHL:	Logical shift left
>> or :SHR:	>> or :SHR:	Logical shift right
Addition, Subtraction, Logical and Boolean Operators		
+	+	Addition
-	-	Subtraction
:LAND: && or :AND:	&& or :LAND: & or :AND:	Logical AND Bitwise AND
:LOR:	|| or :LOR:	Logical OR
|| or :OR:	| or :OR:	Bitwise OR
:LEOR:	XOR or :LEOR:	Logical exclusive OR
^ or :EOR:	^ or :EOR:	Bitwise exclusive OR
Relational or Comparison Operators		
= or ==	= or ==	Equal
/=, <> or !=	<> or !=	Not equal
>	>	Greater than
<	<	Less than
>=	>=	Greater than or equal
<=	<=	Less than or equal
No equivalent	UGT	Unsigned greater than
No equivalent	ULT	Unsigned less than

8. Binary operators in RealView Developer Suite and IAR Embedded Workbench

ASSEMBLER DIRECTIVES

The following table shows the equivalent common assembler directives in RealView Developer Suite and IAR Embedded Workbench.

RealView	IAR	Description
ALIGN	ALIGNROM	Aligns the current location to a specified boundary by padding with zeroes. Note that in IAR Embedded Workbench, there is also a directive called ALIGNRAM that aligns the location counter by incrementing it.
AREA	RSEG	Instructs assembler to assemble a new code or data section
CODE16	CODE16	Instructs assembler to interpret subsequent instructions as 16-bit Thumb instructions
CODE32 or ARM	CODE32	Instructs assembler to interpret subsequent instructions as 32-bit ARM instructions
DATA	DATA	Defines an area of data within a code segment. Note that in RealView Developer Suite, this directive is no longer needed and is ignored by the assembler.
DCB or =	DCB or DC8	Allocates one or more bytes of memory and defines initial runtime contents of the memory
DCD or &	DCD or DC32	Allocates one or more words of memory, aligned on 4-byte boundaries and defines initial runtime contents of the memory
DCFD	DF64	Allocates memory for word-aligned double-precision floating-point numbers and defines initial runtime contents of the register
DCW	DCW or DC16	Allocates one or more half words of memory, aligned on 2-byte boundaries and defines the initial runtime contents of the memory
END	END	Informs the assembler that the end of a source file has been reached
ENTRY	END *expression*	Declares an entry point to a program. In IAR Embedded Workbench, *expression* provides the entry point address. An entry point to a program can also be defined with the linker command line option -s in IAR Embedded Workbench
EQU or *	EQU or =	Gives a symbolic name to a numeric constant, a register-relative value or a program-relative value

RealView	IAR	Description
EXPORT or GLOBAL	EXPORT or PUBLIC	Declares a symbol that can be used by the linker to resolve symbol references in separate object and library files. Note that in IAR Embedded Workbench, #include may also be used.
INCLUDE or GET	INCLUDE or $	Includes a file within the file being assembled
INCBIN	No equivalent	Includes a binary file as it is (without being assembled) within the file being assembled. There is no direct equivalent in IAR Embedded Workbench, but can be defined with the linker command line option --image_input
IMPORT	IMPORT or EXTERN	Provides the assembler with a name that is not defined in the current assembly
LTORG	LTORG	Instructs the assembler to assemble the current literal pool immediately following the directive
RN	No equivalent	Defines a register name for a specified register
SPACE or %	No equivalent	Reserves a zeroed block of memory. There is no direct equivalent in IAR Embedded Workbench, but a workaround to this would be to use the REPT directive to zero a block of memory. Alternatively, the DS8, DS16, DS24, or DS32 directives may be used, but the memory is not filled with zeroes. If these directives are used, the default ROM/Flash content will be preserved.

9. Assembler directives in RealView Developer Suite and IAR Embedded Workbench

The example below compares the use of directives in RealView Developer Suite and IAR Embedded Workbench.

RealView		IAR		Description/Comments
	INCLUDE “header.inc”		INCLUDE “header.inc”	;Include a header file
data	INCBIN “data.dat”		.	;Include a binary file.
	.		.	
	AREA *fred*,CODE		RSEG *fred*:CODE:NOROOT(2)	;Assembles a new code section called *fred*
	ENTRY		.	;Entry point to the program.
	CODE32		CODE32	;Following instructions are 32-bit ARM instructions
	BX func		BX func	;Branch and change to Thumb state
	.		.	
	.		.	
	CODE16		CODE16	;Following instructions are 16-bit Thumb instructions
	BX thumb		BX thumb	;Branch and change back to ARM state
	.		.	
	.		.	
	AREA *john*,DATA		RSEG *john*:DATA:NOROOT(2)	;Assembles a new data section called *john*
	.		.	
	ALIGN 16		ALIGNROM 4	;Aligns current location to 16-byte boundaries
table	DCB “test”	*table*	DC8 'test'	;Defines a string
	DCD 1,5,10		DC32 1,5,10	;Defines 3 words containing decimal values 1, 5 and 10
	DCFD 1.2E-8		DF64 1.2E-8	;Defines a floating point number 1.2×10^{-8}
	DCW -255		DC16 -225	;Defines a halfword with a value of -255
test	EQU 5	*test*	EQU 5	;Assign *test* a value of 5
	.		.	
tab	RN 4		.	;Defines *tab* for register 4
	ADR *tab*,*table*		ADR r4,*table*	;Load address of *table* into register 4
	ADRL *tab*,*table*		ADRL r4,*table*	;Load address of *table* into register 4
	LDR r0,=*table*		LDR r0,=*table*	;Load address of *table* into register 0
	LTORG		LTORG	;Assemble current literal pool
	SPACE 50		DC8 0x32	;Reserves 50 bytes of memory
	EXPORT *table*		EXPORT *table*	;Export the label *table*
	.		.	
	END		END	;End of source file

10. Use of directives in RealView Developer Suite and IAR Embedded Workbench

CONVERTING PSEUDO-INSTRUCTIONS

The following table compares the available pseudo-instructions on RealView Developer Suite and equivalent instructions on IAR Embedded Workbench.

RealView	IAR	Mode	Description
ADR	ADR	ARM, Thumb	Load a program-relative or register-relative address into a register (short range)
ADRL	ADRL	ARM	Load a program-relative or register-relative address into a register (wide range)
LDR	LDR	ARM, Thumb	Load a register with a 32-bit constant value or an address
NOP	NOP	ARM, Thumb	Generate the preferred ARM no-operation code
MOV	MOV	Thumb	Move the value of a low register to another low register (R0–R7). This translates to the instruction: ADD R*n*,R*n*,0
No equivalent	BLF	ARM, Thumb	Calls functions that may be far away or in ARM/Thumb mode

11. Pseudo-instructions in RealView Developer Suite and IAR Embedded Workbench

ASSEMBLER DIFFERENCES

This section highlights other differences between and the RealView ARM Assembler, `armasm` and the ARM IAR Embedded Workbench Assembler, `aarm`.

Label differences

In both RealView Developer Suite and IAR Embedded Workbench, symbols representing addresses or memory locations of instructions or data are referred to as labels. Labels can be program-relative, register-relative, or absolute. There are no label differences between RealView Developer Suite and IAR Embedded Workbench.

Symbol naming rules

In RealView Developer Suite and IAR Embedded Workbench, user-defined symbols can use `a` to `z` (lowercase letters), `A` to `Z` (uppercase letters), `0` to `9` (numeric characters) or `_` (underscore). Numeric characters cannot be used for the first character of symbol names, although in IAR Embedded Workbench the `?` (question mark) may be used to begin a symbol name and the `$` (dollar) may also be included in a symbol name. User-defined symbols in IAR Embedded Workbench can be up to 255 characters long.

Symbol names are case-sensitive, all character names in the symbol are significant and the symbol name must be unique. For built-in symbols such as instructions, registers, operators, and directives, case is insignificant.

Symbols are allowed to contain any printable characters if they are delimited with the `|` (single bar) in RealView Developer Suite or the `` ` `` (backquote) in IAR Embedded Workbench. Note that the single bars or backquotes do not form part of the symbol.

The examples below define the symbol `#funny-label@`:

RealView: `|#funny-label@|`
IAR: `` `#funny-label@` ``

Numeric literals

Numeric literals in RealView Developer Suite and IAR Embedded Workbench can be of the binary, octal, decimal, hexadecimal, character or floating-point type. The table below shows the examples of the forms taken by numeric literals in RealView Developer Suite and IAR Embedded Workbench.

Type	RealView Examples	IAR Examples
Binary	No equivalent	0101b, b'0101'
Octal	No equivalent	1234q, q'1234'
Decimal	1234, -1234	1234, -1234, d'1234'
Hexadecimal	0xFFFF, &FFFF	0xFFFF, 0FFFFh, h'FFFF'
ASCII character	'ABCD'	'ABCD'
Floating-point	12.3, 1.23E-24, -1.23e-24, 1.0E3	12.3, 1.23E-24, -1.23e-24, 1.0E3

12. Numeric literals in RealView Developer Suite and IAR Embedded Workbench

Numeric expressions

In both RealView Developer Suite and IAR Embedded Workbench, numeric expressions consist of combinations of numeric constants, numeric variables, ordinary numeric literals, binary operators, and parentheses. Numeric expressions evaluate to 32-bit integers, which have an unsigned range from 0 to 2^{32} - 1 and a signed range from -2^{31} to 2^{31} -1.

SPECIFIC DIRECTIVES REFERENCE

This section describes some of the more complex assembler directives available in RealView Developer Suite and how to change them to work with IAR Embedded Workbench .

AREA directive

In RealView Developer Suite, the AREA directive instructs the assembler to assemble a new code or data section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker.

In IAR Embedded Workbench, the equivalent directive is the RSEG directive. The RSEG directive is used to begin a program.

Syntax

RealView Developer Suite:

```
AREA sectionname{,attr}{,attr}...
```

Where *sectionname* = the name to be given to the section

attr = one or more comma-delimited section attributes. Valid attributes include ALIGN=*expression*, ASSOC=*section*, CODE, COMDEF, COMMON, DATA, NOALLOC, NOINIT, READONLY, READWRITE

IAR Embedded Workbench:

```
RSEG segmentname [:type][flag][(align)]
```

where *segmentname* = the name assigned to the segment

type = the memory type, typically CODE or DATA (and types supported by the IAR XLINK Linker)

flag = may either be NOROOT, REORDER, or SORT. NOROOT indicates that the segment part may be discarded by the linker even if no symbols in this segment are referred to. All segment parts except startup code and interrupt vectors should set this flag. The default mode is ROOT, which indicates that the segment part must not be discarded. REORDER allows the linker to reorder segment parts. The default mode is NOREORDER, which indicates that the segment parts must remain in order. SORT allows the linker to sort the segment parts in decreasing alignment order. The default mode is NOSORT which indicates that the segment parts will not be sorted.

align = exponent of the value to which the address should be aligned, in the range of 0 to 30. For example, if *align* is 1, this results in word alignment 2

Example

RealView Developer Suite:

The following example defines a read-only code section named Test.

```
AREA Test,CODE,READONLY
```

IAR Embedded Workbench:

The following example defines a 32-bit code segment named Test.

```
RSEG Test:CODE:NOROOT(2)
```

MAP directive

In RealView Developer Suite, the MAP directive sets the origin of a storage map to a specified address. This directive is used in conjunction with the FIELD directive to describe a storage map.

In IAR Embedded Workbench, there is no equivalent directive.

Syntax

RealView Developer Suite:

```
MAP expr{,base-register}
```

Where *expr* = numeric or program-relative expression

base-register = specifies a register. If specified, the address where the storage map starts is the sum of *expr* and the value of *base-register* at runtime

IAR Embedded Workbench:

In IAR Embedded Workbench, there is no equivalent directive. See the `FIELD` directive below below for how to convert this construct.

Example

RealView Developer Suite:

The following example shows that the storage maps starts at the address stored in register `r9`.

```
  MAP  0,r9
```

IAR Embedded Workbench:

In IAR Embedded Workbench, there is no equivalent example.

FIELD directive

In RealView Developer Suite, the `FIELD` directive describes space within a storage map that has been defined using the `MAP` directive.

In IAR Embedded Workbench, there is no equivalent directive, although the `EQU` directive may be used to achieve the same purpose.

Syntax

RealView Developer Suite:

`{`*`label`*`}` `FIELD` *`expr`*

Where *`label`* = optional label. If specified, *`label`* is assigned the value of storage location counter

`expr` = expression that evaluates to the number of bytes to increment the storage counter

IAR Embedded Workbench:

`Label` `EQU` *`expr`*

where *`label`* = symbol to be defined

`expr` = value assigned to symbol

Example

RealView Developer Suite:

The following example shows how the `MAP` and `FIELD` directives are used to define register-relative labels:

```
        MAP     0,r9    ;Set storage location counter to address stored in r9
        FIELD   8       ;Increment storage location counter by 8 bytes
Code    FIELD   4       ;Set Code to the address [r9 + 8] and increment storage
                        ;location counter by 4 bytes
Size    FIELD   4       ;Set Size to the address [r9 + 12] and increment storage
                       ;location counter by 4 bytes
  .
  .
        MOV     r9,...
        LDR     r0,Code ;Equivalent to LDR r0,[r9,#8]
```

IAR Embedded Workbench:

The following example shows the equivalent instructions in IAR to define register-relative labels:

```
Code    EQU     8       ;Set Code to the address [r9 + 8]
Size    EQU     12      ;Set Size to the address [r9 + 12]
  .
  .
        MOV     r9,...
        LDR     r0,[r9,#Code]
```

Advanced conversion

PREDEFINED SYMBOLS

The following table compares the predefined symbols available in RealView Developer Suite and IAR Embedded Workbench.

RealView	IAR	Description
{ARCHITECTURE}	No equivalent	Name of selected ARM architecture
{AREANAME}	No equivalent	Name of the current AREA
{ARMASM_VERSION} or |ads$version|	__VER__	Integer that increases with each version number
{CODESIZE} or {CONFIG}	No equivalent	Has the value 32 if assembler is assembling ARM code, or 16 if assembling Thumb code
{COMMANDLINE}	No equivalent	Holds the contents of the command line
{CPU}	No equivalent	Name of selected CPU
{ENDIAN}	__BIG_ENDIAN__ or __LITTLE_ENDIAN__	In RealView Developer Suite, the value 'big' or 'little' is returned depending on the assembler mode. In IAR Embedded Workbench, the symbol expands to the number 1 when the code is compiled, thereby identifying the byte order in use
{FPU}	No equivalent	Name of selected fpu
{INPUTFILE}	__FILE__	String indicating the name of the current source file
{INTER}	No equivalent	Has the value True if /inter is set. The default is False
{LINENUM}	__LINE__	Integer indicating line number in current source file
{NOSWST}	No equivalent	Has the value True if /swst is set. The default is False
{OPT}	No equivalent	Holds the value of the currently set listing option
{PC} or .	.	Address of current instruction
{PCSTOREOFFSET}	No equivalent	Offset between the address of the STR pc, [...] or STM Rb,{...,pc} instruction and the value of pc stored out
{ROPI}	No equivalent	Has the value True if /ropi is set. The default is False
{RWPI}	No equivalent	Has the value True if /rwpi is set. The default is False
{SWST}	No equivalent	Has the value True if /swst is set. The default is False
{VAR} or @	No equivalent	Current value of storage area location counter
No equivalent	__DATE__	String in dd/mm/yyyy format indicating the current date
No equivalent	__IAR_SYSTEMS_ASM__	Hold the IAR Embedded Workbench assembler identifier
No equivalent	__TID__	Target identity consisting of 2-bytes. High byte is target identity, 0x49 for AARM), low byte is unused
No equivalent	__TIME__	String in hh:mm:ss format indicating current time

13. Predefined symbols in RealView Developer Suite and IAR Embedded Workbench

CONDITIONAL ASSEMBLY

The following table shows the equivalent conditional assembly directives in RealView Developer Suite and IAR Embedded Workbench.

RealView	IAR	Description
IF or [	IF	Assemble a sequence of instructions if condition is true
ELSE or |	ELSE	Assemble a sequence of instructions if condition is false
ENDIF or]	ENDIF	Marks the end of a sequence of instructions that were conditionally assembled
ELIF	ELSEIF	Creates a structure equivalent to ELSE IF, without the nesting or repeating the condition
WHILE	REPT	Begins a sequence of instructions that are assembled repeatedly.
WEND	ENDR	Terminates a sequence of instructions that are assembled repeatedly
INCLUDE, GET or #include	INCLUDE, $ or #include	Includes a file within the file being assembled. In RealView Developer Suite, #include may be used if the file is preprocessed with the C preprocessor, before using armasm to assemble it.
SETA	SETA, ASSIGN, VAR or #define	Sets the value of a local or global arithmetic variable
No equivalent	#error	Generates an error
No equivalent	#message	Generate message on standard output
[:DEF: *symbol*	#ifdef	Assemble a sequence of instructions if symbol is defined
[:NOT: :DEF: *symbol*	#ifndef	Assemble a sequence of instructions if symbol is undefined
No equivalent	#undef	Undefine a label

14. Conditional assembly directives in RealView Developer Suite and IAR Embedded Workbench

The example below compares the use of conditional assembly directives in RealView Developer Suite and IAR Embedded Workbench. It defines two different options for a FFT routine (1,2) plus an option with no routine.

RealView	IAR	Description/Comments
`FFT_VARIANT SETA 1`	`#define FFT_VARIANT 1`	`;Define a variable called FFT_VARIANT that has a value of 1`
`[ DUMMY = 1`	`IF DUMMY == 1`	`;Assemble sequence of instructions as condition is true`
`fft`	`fft`	
`MOV R0,#10`	`MOV R0,#10`	`;Set up R0`
`.`	`.`	
`.`	`.`	
`MOV PC,LR`	`MOV PC,LR`	`;Return`
`ELIF FFT_VARIANT = 2`	`ELSEIF FFT_VARIANT == 2`	`;Assemble sequence of instructions if condition is false`
`fft`	`fft`	`;FFT type 1`
`.`	`.`	
`.`	`.`	
`|`	`ELSE`	`;Assemble sequence of instructions if the next condition is true`
`fft`	`fft`	`;FFT type 2`
`MOV R0, #-1`	`MOV R0, #-1`	`;Set up R0 (no fft available)`
`MOV PC,LR`	`MOV PC,LR`	`;Return`
`]`	`ENDIF`	`;End of conditionally assembled instructions`

15. Use of conditional assembly directives in RealView Developer Suite and IAR Embedded Workbench

MACROS

Macros are user-defined symbols that represent a block of one or more assembler source lines. The symbol can then be used instead of repeating the whole block of code several times. The following table shows the equivalent macro processing directives in RealView Developer Suite and IAR Embedded Workbench.

RealView	IAR	Description
`MACRO`	`MACRO`	Define the start of a macro
`MEND`	`ENDM`	Define the end of a macro
`MEXIT`	`EXITM`	Generate premature exit from a macro
`LCLA, LCLL or LCLS`	`LOCAL`	Create symbols local to a macro. In RealView Developer Suite, `LCLA` declares an arithmetic value (initialized to `0`), `LCLL` declares a logical variable (initialized to `{FALSE}`) and `LCLS` declares a string variable (initialized to a null string, `" "`)

16. Macro processing directives in RealView Developer Suite and IAR Embedded Workbench

The example below compares the use of macro processing directives in RealView Developer Suite and IAR Embedded Workbench for decrementing a variable .

RealView		IAR		Descriptions
	`AREA` *`count`*`,CODE`		`RSEG` *`count`*`:CODE:NOROOT(2)`	`;Assemble the source file` *`count`*
	`ENTRY`		`.`	
	`.`		`.`	
	`MACRO`	*`countdown`*	`MACRO` *`start`*	`;Start of macro called` *`countdown`*
`$label`	`countdown $start`		`.`	`;Parameter accepted by the macro`
	`LCLA` *`value`*		`LOCAL` *`value`*	`;Create a local symbol`
`value`	`SETA $start`	*`value`*	`SETA` *`start`*	`;Assign` *`value`* `the value of` *`start`*
	`WHILE value > 0`		`REPT value`	`;Start of repeated statements`
	`DCD` *`value`*		`DC32` *`value`*	`;Define a word called` *`value`*
`value`	`SETA` *`value`* `- 1`	*`value`*	`SETA` *`value`* `- 1`	`;Decrement` *`value`* `by 1`
	`WEND`		`ENDR`	`;End of repeated statements`
	`DCD` *`value`*		`DC32` *`value`*	`;Define a word called` *`value`*
	`MEND`		`ENDM`	`;End of a macro`
	`.`		`.`	
`tab5`	`countdown 5`	`tab5`	`countdown 5`	`;Begin countdown from 5`
	`.`		`.`	
	`END`		`END`	`;End of source file`

17. Use of macro processing directives in RealView Developer Suite and IAR Embedded Workbench

The following list files show the value of *value* counting down from 5 to 1.

RealView Developer Suite listing

```
ARM Macro Assembler    Page 1
    1 00000000                  AREA            test,CODE
    2 00000000                  ENTRY
    3 00000000
    4 00000000                  MACRO
    5 00000000          $label  countdown       $start
    6 00000000                  LCLA            value
    7 00000000
```

```
    8 00000000         value   SETA            $start
    9 00000000
   10 00000000                 WHILE           value > 0
   11 00000000                 DCD             value
   12 00000000         value   SETA            value - 1
   13 00000000                 WEND
   14 00000000                 DCD             value
   15 00000000                 MEND
   16 00000000
   17 00000000         tab50   countdown       5
    6 00000000                 LCLA            value
    7 00000000
    8 00000000 00000005
                       value   SETA            5
    9 00000000
   10 00000000                 WHILE           value > 0
   11 00000000 00000005        DCD             value
   12 00000004 00000004
                       value   SETA            value - 1
   13 00000004                 WEND
   10 00000004                 WHILE           value > 0
   11 00000004 00000004        DCD             value
   12 00000008 00000003
                       value   SETA            value - 1
   13 00000008                 WEND
   10 00000008                 WHILE           value > 0
   11 00000008 00000003        DCD             value
   12 0000000C 00000002
                       value   SETA            value - 1
   13 0000000C                 WEND
   10 0000000C                 WHILE           value > 0
   11 0000000C 00000002        DCD             value
   12 00000010 00000001
                       value   SETA            value - 1
   13 00000010                 WEND
   10 00000010                 WHILE           value > 0
   11 00000010 00000001        DCD             value
   12 00000014 00000000
                       value   SETA            value - 1
   13 00000014                 WEND
   10 00000014                 WHILE           value > 0
   14 00000014 00000000        DCD             value
   18 00000018
   19 00000018                 END

Command Line: --list=test count.s
```

IAR Embedded Workbench listing

```
##############################################################################
#                                                                            #
#     IAR Systems ARM Assembler V4.20A/W32 dd/Mmm/yyyy  hh:mm:ss             #
#     Copyright 1999-2005 IAR Systems. All rights reserved.                  #
#                                                                            #
#           Source file   =  count.s                                         #
#           List file     =  test.lst                                        #
#           Object file   =  count.r79                                       #
#           Command line  =  -l test count.s                                 #
#                                                                            #
##############################################################################

    1    00000000                       RSEG count:CODE:NOROOT(2)
    2    00000000
    2.1  00000000                       ALIGNROM 2
    3    00000000
   17    00000000
   18    00000000              tab5     countdown       5
   18.1  00000000
   18.2  00000000                       LOCAL   value
   18.3  00000000
   18.4  00000005              value    SETA    5
   18.5  00000000                       REPT    value
   18.6  00000000                       DC32    value
   18.7  00000000              value    SETA    value - 1
```

```
   18.8  00000000
   18.9  00000000                      ENDR
   18    00000000 05000000            DC32    value
   18.1  00000004              value   SETA    value - 1
   18.2  00000004
   18    00000004 04000000            DC32    value
   18.1  00000003              value   SETA    value - 1
   18.2  00000008
   18    00000008 03000000            DC32    value
   18.1  00000002              value   SETA    value - 1
   18.2  0000000C
   18    0000000C 02000000            DC32    value
   18.1  00000001              value   SETA    value - 1
   18.2  00000010
   18    00000010 01000000            DC32    value
   18.1  00000000              value   SETA    value - 1
   18.2  00000014
   18    00000014 00000000    tab5    countdown       5
   18.1  00000018 0000A0E1            NOP
   18.2  0000001C                     ENDM
   19    0000001C
   20    0000001C                     END
##############################
#          CRC:4E7E          #
#        Errors:   0         #
#        Warnings: 0         #
#         Bytes: 28          #
##############################
```

MODULES

In IAR Embedded Workbench, module directives are used to create libraries containing many small modules, where each module represents a single routine. The number of source and object files can be reduced using module directives. There is no direct equivalent in RealView Developer Suite, but a similar result can be achieved using the AREA directive.

RealView	IAR	Description
No equivalent	MODULE or LIBRARY	Defines the beginning of a library module.
No equivalent	ENDMOD	Defines the end of a library module

The Call Frame Information (CFI) directives are used to define backtrace information for the instructions in a program. The backtrace information is used to keep track of the contents of resources in the assembler code. In the case of library functions and assembler code, backtrace information has to be added in order to use the call frame stack in the debugger.

Linker and other tools

In ARM RealView Developer Suite, the linker is called `armlink`, while in the IAR Embedded Workbench IDE, the linker is called the IAR XLINK Linker.

LINKER COMMAND LINE OPTIONS

The table below compares the basic linker command line options in RealView Developer Suite and IAR Embedded Workbench.

RealView	IAR	Description
`--help or -h`	No equivalent	Prints summary of commonly used command line options. There is no direct equivalent in IAR Embedded Workbench, but the task can be performed by invoking `xlink` without arguments.
`--vsn`	No equivalent	Displays `armlink` version information and license details. There is no direct equivalent in IAR Embedded Workbench, but the task can be performed by invoking `xlink` without arguments.
`--ro-base or -ro` *address*	`-Z` *type segment*`=start`	Sets load and execution addresses of the region containing the read-only output section
`--rw-base or -rw` *address*	`-Z` *type segment=start*	Sets execution addresses of the region containing the read-write output section
`--first` *section-id*	`-Z` *segment=start-end* or `-Z` *segment=start*`:+`*size*	Places the selected input section first in its execution region
`--last` *section-id*	`-Z` *segment=start-end* or `-Z` *segment=start*`:+`*size*	Places the selected input section last in its execution region
`--entry` *location*	`-s` *symbol*	Specifies the unique entry point of the image
`--libpath` *pathlist*	`-I`*pathname*	Specifies a list of paths used to search for ARM standard C/C++ libraries
`--userlibpath` *pathlist*	`-I`*pathname*	Specifies a list of paths used to search for user libraries
`--remove`	No equivalent	Removes unused sections from the image. This is performed by default in IAR Embedded Workbench.

RealView	IAR	Description
`--map`	`-l file -xm`	Creates an image/module map.
`--symbols or -s`	`-l file -xe`	Lists all local and global symbols used in linking, and their values
`--xref`	`-l file -xm`	Lists all cross-references between input sections
`--list file`	`-l file`	Redirects the diagnostics from the output of the command line options to a file
`--verbose or -v`	No equivalent	Prints detailed information about the link operation, including objects and libraries
`--via file`	`-f file`	Reads a list of input filenames and linker options from a file
`--output or -o file`	`-o file`	Specifies the name of the output file

18. Linker command line options in RealView Developer Suite and IAR Embedded Workbench

LINKER SCATTER LOADING AND SEGMENT CONTROL

In order to specify the memory map of an image to the linker, RealView Developer Suite utilizes the scatter loading mechanism. Although there is no direct IAR Embedded Workbench equivalent to this mechanism, a similar result can be achieved through segment control.

With RealView Developer Suite, depending on the complexity of the memory maps of the image, images that have simple memory maps may also be created using command line options. Scatter loading is used for images that have a complex memory map where complete control is required over the grouping and placement of image components, for example, in situations where there are different types of memory or memory-mapped I/O. The command line option for scatter loading in RealView Developer Suite is:

`--scatter filename`

This option instructs the linker to construct the image memory map as described in the description file *`filename`*.

As mentioned previously, the linker in IAR Embedded Workbench does not have a single equivalent command line option. However, a similar result can be achieved with segment control using multiple `-Z` options to allocate or place segments in memory. Segment placement is performed one placement command at a time, taking in to account previous placement commands. As each command is processed, any parts of the ranges given for that placement command that are already in use (for example, by segments placed with earlier segment placement commands) are removed from the considered ranges.

Furthermore, the `-Q` option in IAR Embedded Workbench can be used to do automatic setup for copy initialization of segments. The command line option has the format below:

`-Qsegment=initializer_segment`

This option will make the linker place all data contents of the segment *`segment`* into a segment *`initializer_segment`*. Debugging information, etc, is still associated with the segment *`segment`*. At runtime, the application must copy the contents of *`initializer_segment`* (in ROM) to *`segment`* (in RAM) using any suitable method of copy (the standard `memcpy` routine is perhaps the easiest way). This is useful for code that needs to be in RAM.

The table below shows an example of a simple RealView Developer Suite scatter loading description file for loading code and data sections into non-contiguous regions in memory. The description file loads code (`RO`) at address `0x0000` in memory, data (`RW`) at address `0xA000` in memory, and dynamically creates a zero-initialized (`ZI`) section at runtime.

Description File Listing	Description/Comments
`LR_1 0x0000`	`;Define load region LR_1`
`{`	
`  ER_RO +0`	`;The execution region containing code, ER_RO has no offset and begins at address 0x0000`
`  {`	
`    * (+RO)`	`;All RO sections are placed consecutively into this region`
`  }`	
`  ER_RW 0xA000`	`;The execution region containing data, ER_RW is offset to address 0xA000`
`  {`	
`    * (+RW)`	`;All RW sections are placed consecutively into this region`
`  }`	
`  ER_ZI +0`	`;The execution region containing the ZI section, ER_ZI has no offset and is placed at address 0xA000 + size of the ER_RW region`
`  {`	
`    * (+ZI)`	`;All ZI sections are placed consecutively into this region`
`  }`	
`}`	

19. Example of scatter loading and segment control

Note that the equivalent linker command line option in RealView Developer Suite is:

```
armlink --ro-base 0x0000 --rw-base 0xA000
```

The equivalent segment placement commands in IAR Embedded Workbench for placing a code segment at address `0x0000` in memory and a data segment at address `0xA000` in memory are the following:

```
-Z (CODE) SEG_RO = 0x0000
-Z (DATA) SEG_RW,SEG_ZI = 0xA000
```

In RealView Developer Suite, function type attributes can be specified either before or after the return type:

```
__irq void InterruptHandler (void);
void __irq InterruptHandler (void);
```

In IAR Embedded Workbench, function type attributes can only be specified *before* the return type:

```
__irq void InterruptHandler (void);
```