SAM8 IAR Assembler

Reference Guide

for Samsung's

SAM8 Microcontroller Family

EDITION NOTICE

Second edition: April 2003

Part number: ASAM8-2

Contents

Tables

Preface

Welcome to the SAM8 IAR Assembler Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use the SAM8 IAR Assembler to best suit your application requirements.

Who should read this guide

You should read this guide if you plan to develop an application using assembler language for the SAM8 microcontroller and need to get detailed reference information on how to use the SAM8 IAR Assembler. In addition, you should have working knowledge of the following:

- The architecture and instruction set of the SAM8 microcontroller. Refer to the documentation from Samsung for information about the SAM8 microcontroller
- General assembler language programming
- Application development for embedded systems
- The operating system of your host machine.

How to use this guide

When you first begin using the SAM8 IAR Assembler, you should read the *Introduction to the SAM8 IAR Assembler* chapter in this reference guide.

If you are an intermediate or advanced user, you can focus more on the reference chapters that follow the introduction.

If you are new to using the IAR toolkit, we recommend that you first read the initial chapters of the *SAM8 IAR Embedded Workbench™ IDE User Guide*. They give product overviews, as well as tutorials that can help you get started.

What this guide contains

Below is a brief outline and summary of the chapters in this guide.

- *Introduction to the SAM8 IAR Assembler* provides programming information. It also describes the source code format, and the format of assembler listings.
- *Assembler options* first explains how to set the assembler options from the command line and how to use environment variables. It then gives an alphabetical summary of the assembler options, and contains detailed reference information about each option.

- *Assembler operators* gives a summary of the assembler operators, arranged in order of precedence, and provides detailed reference information about each operator.
- *Assembler directives* gives an alphabetical summary of the assembler directives, and provides detailed reference information about each of the directives, classified into groups according to their function.
- *Diagnostics* contains information about the formats and severity levels of diagnostic messages.

Other documentation

The complete set of IAR Systems development tools for the SAM8 microcontroller is described in a series of guides. For information about:

- Using the IAR Embedded Workbench™ and the IAR C-SPY™ Debugger, refer to the *SAM8 IAR Embedded Workbench™ IDE User Guide*
- Programming for the SAM8 IAR C Compiler, refer to the *SAM8 IAR C Compiler Reference Guide*
- Using the IAR XLINK Linker™, the IAR XLIB Librarian™, and the IAR XAR Library Builder™, refer to the *IAR Linker and Library Tools Reference Guide*.
- Using the IAR C Library, refer to the *IAR C Library Functions Reference Guide*, available from the IAR Embedded Workbench IDE **Help** menu.

All of these guides are delivered in PDF format on the installation media. Some of them are also delivered as printed books.

Document conventions

This guide uses the following typographic conventions:

Style	Used for
`computer`	Text that you enter or that appears on the screen.
`parameter`	A label representing the actual value you should enter as part of a command.
`[option]`	An optional part of a command.
`{a | b | c}`	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
reference	A cross-reference within or to another part of this guide.

Table 1: Typographic conventions used in this guide

Style	Used for
	Identifies instructions specific to the versions of the IAR Systems tools for the IAR Embedded Workbench interface.
	Identifies instructions specific to the command line versions of IAR Systems development tools.

Table 1: Typographic conventions used in this guide (Continued)

Introduction to the SAM8 IAR Assembler

This chapter describes the source code format for the SAM8 IAR Assembler and provides programming hints.

Refer to Samsung's hardware documentation for syntax descriptions of the instruction mnemonics.

Source format

The format of an assembler source line is as follows:

[*label* [:]] [*operation*] [*operands*] [; *comment*]

where the components are as follows:

label	A label, which is assigned the value and type of the current program location counter (PLC). The `:` (colon) is optional if the label starts in the first column.
operation	An assembler instruction or directive. This must not start in the first column.
operands	An assembler instruction can have zero, one, or more operands.
	The data definition directives, for example DB and DC8, can have any number of operands. For reference information about the data definition directives, see *Space allocation directives*, page 74.
	Other assembler directives can have one, two, or three operands, separated by commas.
comment	Comment, preceded by a `;` (semicolon).

The fields can be separated by spaces or tabs.

A source line may not exceed 2047 characters.

Tab characters, ASCII `09H`, are expanded according to the most common practice; i.e. to columns 8, 16, 24 etc.

The SAM8 IAR Assembler uses the default filename extensions `s18`, `asm`, and `msa` for source files.

Assembler expressions

Expressions can consist of operands and operators.

The assembler will accept a wide range of expressions, including both arithmetic and logical operations. All operators use 32-bit two's complement integers, and range checking is only performed when a value is used for generating code.

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators; see also *Precedence of operators*, page 23.

The following operands are valid in an expression:

- User-defined symbols and labels
- Constants, excluding floating-point constants
- The program location counter (PLC) symbol, `$`.

These are described in greater detail in the following sections.

The valid operators are described in the chapter *Assembler operators*, page 23.

TRUE AND FALSE

In expressions a zero value is considered FALSE, and a non-zero value is considered TRUE.

Conditional expressions return the value 0 for FALSE and 1 for TRUE.

USING SYMBOLS IN RELOCATABLE EXPRESSIONS

Expressions that include symbols in relocatable segments cannot be resolved at assembly time, because they depend on the location of segments.

Such expressions are evaluated and resolved at link time, by the IAR XLINK Linker™. There are no restrictions on the expression; any operator can be used on symbols from any segment, or any combination of segments.

For example, a program could define the segments `DATA` and `CODE` as follows:

```
        NAME    prog1
        EXTERN  third
        RSEG    DATA
first:  DC8     5
second: DC8     3
        ENDMOD
        MODULE  prog2
        RSEG    CODE
start   …
```

Then in the segment `CODE` the following instructions are legal:

```
LD      R7,first
LD      R7,first+1
LD      R7,1+first
LD      R7,(first/second)*third
```

Note: At assembly time, there will be no range check. The range check will occur at link time and, if the values are too large, there will be a linker error.

SYMBOLS

User-defined symbols can be up to 255 characters long, and all characters are significant.

Symbols must begin with a letter, a–z or A–Z, ? (question mark), or _ (underscore). Symbols can include the digits 0–9 and `$` (dollar).

For built-in symbols like instructions, registers, operators, and directives, case is insignificant. For user-defined symbols case is by default significant but can be turned on and off using the **Case sensitive user symbols** (`-s`) assembler option. See page 19 for additional information.

Note that symbols and labels are byte addresses. For additional information, see *Generating lookup table*, page 75.

LABELS

Symbols used for memory locations are referred to as labels.

Program location counter (PLC)

The program location counter is called `$`. For example:

```
JR   T,$       ; Loop forever
```

INTEGER CONSTANTS

Since all IAR Systems assemblers use 32-bit two's complement internal arithmetic, integers have a (signed) range from -2147483648 to 2147483647.

Constants are written as a sequence of digits with an optional `-` (minus) sign in front to indicate a negative number.

Commas and decimal points are not permitted.

The following types of number representation are supported:

Integer type	Example
Binary	`1010b`, `b'1010'`
Octal	`1234q`, `q'1234'`
Decimal	`1234`, `-1`, `d'1234'`
Hexadecimal	`0FFFFh`, `0xFFFF`, `h'FFFF'`

Table 2: Integer constant formats

Note: Both the prefix and the suffix can be written with either uppercase or lowercase letters.

ASCII CHARACTER CONSTANTS

ASCII constants can consist of between zero and more characters enclosed in single or double quotes. Only printable characters and spaces may be used in ASCII strings. If the quote character itself is to be accessed, two consecutive quotes must be used:

Format	Value
'ABCD'	ABCD (four characters).
"ABCD"	ABCD'\0' (five characters the last ASCII null).
'A''B'	A'B
`'A'''`	`A'`
`''''` (4 quotes)	'
`''` (2 quotes)	Empty string (no value).
""	Empty string (an ASCII null character).
\'	'
\\	\

Table 3: ASCII character constant formats

PREDEFINED SYMBOLS

The SAM8 IAR Assembler defines a set of symbols for use in assembler source files. The symbols provide information about the current assembly, allowing you to test them in preprocessor directives or include them in the assembled code. The strings returned by the assembler are enclosed in double quotes.

The following predefined symbols are available:

Symbol	Value
__ASAM8__	Target identity.
__DATE__	Current date in dd/Mmm/yyyy format (string).
__FILE__	Current source filename (string).
__IAR_SYSTEMS_ASM__	IAR assembler identifier (number).
__LINE__	Current source line number (number).

__TID__	Target identity, consisting of two bytes. The high byte is the target identity, which is `8` for `ASAM8`. Bits 7-4 of the low byte give the Processor configuration option, and bits 3-0 give the Default data pointer size option.
	The following values are therefore possible:
	`-v0` `-ut` `0x0800`
	`-v0` `-un` `0x0801`
	`-v1` `-ut` `0x0810`
	`-v1` `-un` `0x0811`
	`-v2` `-ut` `0x0820`
	`-v2` `-un` `0x0821`
	`-v3` `-ut` `0x0830`
	`-v3` `-un` `0x0831`
	`-v4` `-ut` `0x0840`
	`-v4` `-un` `0x0841`

__TIME__	Current time in `hh:mm:ss` format (string).
__VER__	Version number in integer format; for example, version 4.17 is returned as 417 (number).

Table 4: Predefined symbols

Notice that `__TID__` is related to the predefined symbol `__TID__` in the SAM8 IAR C Compiler. It is described in the *SAM8 IAR C Compiler Reference Guide*.

Including symbol values in code

To include a symbol value in the code, you use the symbol in one of the data definition directives.

For example, to include the time of assembly as a string for the program to display:

```
timdat DC8    __TIME__,",",__DATE__,0 ; time and date
       ...
       LD     RR4,#timdat             ;  load address of string
       CALL   printstring             ;  routine to print string
```

Testing symbols for conditional assembly

To test a symbol at assembly time, you use one of the conditional assembly directives.

For example, in a source file written for use on any one of the SAM8 family members, you may want to assemble appropriate code for a specific microcontroller. You could do this using the `__TID__` symbol as follows:

```
#define TARGET ((__TID__& 0x0F0)>>4)
#if (TARGET==1)
...
...
#else
...
...
#endif
```

Register symbols

The following table shows the existing predefined register symbols:

Name	Address size	Description
R0–R15	8 bits	Byte registers
RR0, RR2, ..., RR14	16 bits	Word registers
PC		Program counter
SP		Stack pointer, word
SPH		Stack pointer, high byte
SPL		Stack pointer, low byte

Table 5: Predefined register symbols

Programming hints

This section gives hints on how to write efficient code for the SAM8 IAR Assembler. For information about projects including both assembler and C source files, see the *SAM8 IAR C Compiler Reference Guide*.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of SAM8 derivatives are included in the IAR product package, in the `\sam8\inc` directory. These header files define the processor-specific special function registers (SFRs) and interrupt vector numbers.

The header files are intended to be used also with the SAM8 IAR C Compiler, `ICCSAM8`, and they are suitable to use as templates when creating new header files for other SAM8 derivatives.

If any assembler-specific additions are needed in the header file, these can be added easily in the assembler-specific part of the file:

```
#ifdef __IAR_SYSTEMS_ASM__
  (assembler-specific defines)
#endif
```

USING C-STYLE PREPROCESSOR DIRECTIVES

The C-style preprocessor directives are processed before other assembler directives. Therefore, do not use preprocessor directives in macros and do not mix them with assembler-style comments.

Assembler options

This chapter first explains how to set the options from the command line, and gives an alphabetical summary of the assembler options. It then provides detailed reference information for each assembler option.

The *SAM8 IAR Embedded Workbench™ IDE User Guide* describes how to set assembler options in the IAR Embedded Workbench, and gives reference information about the available options.

Setting command line options

To set assembler options from the command line, you include them on the command line, after the `asam8` command:

```
asam8 [options] [sourcefile] [options]
```

These items must be separated by one or more spaces or tab characters.

If all the optional parameters are omitted the assembler will display a list of available options a screenful at a time. Press Enter to display the next screenful.

For example, when assembling the source file `power2.s18`, use the following command to generate a list file to the default filename (`power2.lst`):

```
asam8 power2 -L
```

Some options accept a filename, included after the option letter with a separating space. For example, to generate a list file with the name `list.lst`:

```
asam8 power2 -l list.lst
```

Some other options accept a string that is not a filename. This is included after the option letter, but without a space. For example, to generate a list file to the default filename but in the subdirectory named `list`:

```
asam8 power2 -Llist\
```

Note: The subdirectory you specify must already exist. The trailing backslash is required because the parameter is prepended to the default filename.

EXTENDED COMMAND LINE FILE

In addition to accepting options and source filenames from the command line, the assembler can accept them from an extended command line file.

By default, extended command line files have the extension `xcl`, and can be specified using the `-f` command line option. For example, to read the command line options from `extend.xcl`, enter:

```
asam8 -f extend.xcl
```

Error return codes

When using the SAM8 IAR Assembler from within a batch file, you may need to determine whether the assembly was successful in order to decide what step to take next. For this reason, the assembler returns the following error return codes:

Return code	Description
`0`	Assembly successful, warnings may appear
`1`	There were warnings (only if the `-ws` option is used)
`2`	There were errors

Table 6: Assembler error return codes

ASSEMBLER ENVIRONMENT VARIABLES

Options can also be specified using the `ASMSAM8` environment variable. The assembler appends the value of this variable to every command line, so it provides a convenient method of specifying options that are required for every assembly.

The following environment variables can be used with the SAM8 IAR Assembler:

Environment variable	Description
`ASMSAM8`	Specifies command line options; for example: `set ASMSAM8=-L -ws`
`ASAM8_INC`	Specifies directories to search for include files; for example: `set ASAM8_INC=c:\myinc\`

Table 7: Asssembler environment variables

For example, setting the following environment variable will always generate a list file with the name `temp.lst`:

```
ASMSAM8=-l temp.lst
```

For information about the environment variables used by the IAR XLINK Linker and the IAR XLIB Librarian, see the *IAR Linker and Library Tools Reference Guide*.

Summary of assembler options

The following table summarizes the assembler options available from the command line:

Command line option	Description
-B	Macro execution information
-b	Makes a library module
-c{DMEAO}	Conditional list
-D*symbol*[=*value*]	Defines a symbol
-d	Disables matching
-E*number*	Maximum number of errors
-f *filename*	Extends the command line
-G	Opens standard input as source
-I*prefix*	Includes paths
-i	#included text
-L[*prefix*]	Lists to prefixed source name
-l *filename*	Lists to named file
-M*ab*	Macro quote characters
-N	Omits header from assembler listing
-O*prefix*	Sets object filename prefix
-o *filename*	Sets object filename
-p*lines*	Lines/page
-r[e|n]	Generates debug information
-S	Sets silent operation
-s{+|-}	Case-sensitive user symbols
-T	List active lines
-t*n*	Tab spacing
-U*symbol*	Undefines a symbol
-u[t|n]	Default data pointer
-v[0|1|2|3|4]	Processor configuration
-w[*string*][s]	Disables warnings
-X	Includes unreferenced external symbols
-x{DI2}	Includes cross-references

Table 8: Assembler options summary

Descriptions of assembler options

The following sections give full reference information about each assembler option.

-B

`-B`

Use this option to make the assembler print macro execution information to the standard output stream on every call of a macro. The information consists of:

- The name of the macro
- The definition of the macro
- The arguments to the macro
- The expanded text of the macro.

This option is mainly used in conjunction with the list file options `-L` or `-l`; for additional information, see page 15.

This option is identical to the **Macro execution info** option in the **ASAM8** category in the IAR Embedded Workbench.

-b

`-b`

This option causes the object file to be a library module rather than a program module.

By default, the assembler produces a program module ready to be linked with the IAR XLINK Linker. Use the `-b` option if you instead want the assembler to make a library module.

If the `NAME` directive is used in the source (to specify the name of the program module), the `-b` option is ignored, i.e. the assembler produces a program module regardless of the `-b` option.

This option is identical to the **Make a LIBRARY module** option in the **ASAM8** category in the IAR Embedded Workbench.

-c

`-c{DMEAO}`

Use this option to control the contents of the assembler list file. This option is mainly used in conjunction with the list file options `-L` and `-l`; see page 15 for additional information.

The following table shows the available parameters:

Command line option	Description
`-cD`	Disable list file
`-cM`	Macro definitions
`-cE`	No macro expansions
`-cA`	Assembled lines only
`-cO`	Multiline code

Table 9: Conditional list (-c)

This option is related to the **List file** options in the **ASAM8** category in the IAR Embedded Workbench.

-D `D`*`symbol`*`[=`*`value`*`]`

Use this option to define a preprocessor symbol with the name *`symbol`* and the value *`value`*. If no value is specified, 1 is used.

The `-D` option allows you to specify a value or choice on the command line instead of in the source file.

Example

For example, you could arrange your source to produce either the test or production version of your program dependent on whether the symbol `TESTVER` was defined. To do this, use include sections such as:

```
#ifdef  TESTVER
...     ; additional code lines for test version only
#endif
```

Then select the version required in the command line as follows:

Production version: `asam8 prog`
Test version: `asam8 prog -DTESTVER`

Alternatively, your source might use a variable that you need to change often. You can then leave the variable undefined in the source, and use `-D` to specify the value on the command line; for example:

```
asam8 prog -DFRAMERATE=3
```

This option is identical to the **#define** option in the **ASAM8** category in the IAR Embedded Workbench.

-d `-d`

This option disables `#ifdef`, `#endif` matching.

This option is identical to the **Disable #ifdef/#endif matching** option in the **ASAM8** category in the IAR Embedded Workbench.

-E `-E`*`number`*

This option specifies the maximum number of errors that the assembler will report.

By default, the maximum number is 100. The `-E` option allows you to decrease or increase this number to see more or fewer errors in a single assembly.

-f `-f` *`filename`*

Extends the command line with text read from the specified file. Notice that there must be a space between the option itself and the filename.

The `-f` option is particularly useful where there is a large number of options which are more conveniently placed in a file than on the command line itself. For example, to run the assembler with further options taken from the file `extend.xcl`, use:

```
asam8 prog -f extend.xcl
```

-G `-G`

This option causes the assembler to read the source from the standard input stream, rather than from a specified source file.

When `-G` is used, no source filename may be specified.

-I `-I`*`prefix`*

Use this option to specify paths to be used by the preprocessor by adding the `#include` file search prefix *`prefix`*.

By default, the assembler searches for `#include` files only in the current working directory and in the paths specified in the `ASAM8_INC` environment variable. The `-I` option allows you to give the assembler the names of directories where it will also search if it fails to find the file in the current working directory.

Example

Using the options:

```
-Ic:\global\ -Ic:\thisproj\headers\
```

and then writing:

```
#include "asmlib.hdr"
```

in the source, will make the assembler search first in the current directory, then in the directory `c:\global\`, and finally in the directory `c:\thisproj\headers\`.

This option is related to the **#include** option in the **ASAM8** category in the IAR Embedded Workbench.

`-i` `-i`

Includes `#include` files in the list file.

By default, the assembler does not list `#include` file lines since these often come from standard files and would waste space in the list file. The `-i` option allows you to list these file lines.

This option is related to the **#include** option in the **ASAM8** category in the IAR Embedded Workbench.

`-L` `-L[prefix]`

By default the assembler does not generate a list file. Use this option to make the assembler generate one and sent it to file [*prefix*] *sourcename*`.lst`.

To simply generate a listing, use the `-L` option without a prefix. The listing is sent to the file with the same name as the source, but the extension will be `lst`.

The `-L` option lets you specify a prefix, for example to direct the list file to a subdirectory. Notice that you must not include a space before the prefix.

`-L` may not be used at the same time as `-l`.

Example

To send the list file to `list\prog.lst` rather than the default `prog.lst`:

```
asam8 prog -Llist\
```

This option is related to the **Output directories** options in the **General** category in the IAR Embedded Workbench.

`-l` `-l filename`

Use this option to make the assembler generate a listing and send it to the file *`filename`*. If no extension is specified, `lst` is used. Notice that you must include a space before the filename.

By default, the assembler does not generate a list file. The `-l` option generates a listing, and directs it to a specific file. To generate a list file with the default filename, use the `-L` option instead.

This option is related to the **List** options in the **ASAM8** category in the IAR Embedded Workbench.

`-M` `-Mab`

This option sets the characters to be used as left and right quotes of each macro argument to *`a`* and *`b`* respectively.

By default, the characters are `<` and `>`. The `-M` option allows you to change the quote characters to suit an alternative convention or simply to allow a macro argument to contain `<` or `>` themselves.

Example

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

to call a macro print with `>` as the argument.

Note: Depending on your host environment, it may be necessary to use quote marks with the macro quote characters, for example:

```
asam8 filename -M'<>'
```

This option is identical to the **Macro quote chars** option in the **ASAM8** category in the IAR Embedded Workbench.

`-N` `-N`

Use this option to omit the header section that is printed by default in the beginning of the list file.

This option is useful in conjunction with the list file options `-L` or `-l`; see page 15 for additional information.

This option is related to the **List file** option in the **ASAM8** category in the IAR Embedded Workbench.

-O `-O`*`prefix`*

Use this option to set the prefix to be used on the name of the object file. Notice that you must not include a space before the prefix.

By default the prefix is null, so the object filename corresponds to the source filename (unless `-o` is used). The `-O` option lets you specify a prefix, for example to direct the object file to a subdirectory.

Notice that `-O` may not be used at the same time as `-o`.

Example

To send the object code to the file `obj\prog.r18` rather than to the default file `prog.r18`:

```
asam8 prog -Oobj\
```

This option is related to the **Output directories** option in the **General** category in the IAR Embedded Workbench.

-o `-o` *`filename`*

This option sets the filename to be used for the object file. Notice that you must include a space before the filename. If no extension is specified, `r18` is used.

The option `-o` may not be used at the same time as the option `-O`.

Example

For example, the following command puts the object code to the file `obj.r18` instead of the default `prog.r18`:

```
asam8 prog -o obj
```

Notice that you must include a space between the option itself and the filename.

This option is related to the filename and directory that you specify when creating a new source file or project in the IAR Embedded Workbench.

-p `-plines`

The `-p` option sets the number of lines per page to *`lines`*, which must be in the range 10 to 150.

This option is used in conjunction with the list options `-L` or `-l`; see page 15 for additional information.

This option is identical to the **Lines/page** option in the **ASAM8** category in the IAR Embedded Workbench.

-r `-r[e|n]`

The `-r` option makes the assembler generate debug information that allows a symbolic debugger such as C-SPY to be used on the program.

By default, the assembler does not generate debug information, to reduce the size and link time of the object file. You must use the `-r` option if you want to use a debugger with the program.

The following table shows the available parameters:

Command line option	Description
`-re`	Includes the full source file into the object file
`-rn`	Generates an object file without source information; symbol information will be available.

Table 10: Generating debug information (-r)

This option is identical to the **Generate debug info** option in the **ASAM8** category in the IAR Embedded Workbench.

-S `-S`

The `-S` option causes the assembler to operate without sending any messages to the standard output stream.

By default, the assembler sends various insignificant messages via the standard output stream. Use the `-S` option to prevent this.

The assembler sends error and warning messages to the error output stream, so they are displayed regardless of this setting.

-s `-s{+|-}`

Use the `-s` option to control whether the assembler is sensitive to the case of user symbols:

Command line option	Description
`-s+`	Case-sensitive user symbols
`-s-`	Case-insensitive user symbols

Table 11: Controlling case sensitivity in user symbols (-s)

By default, case sensitivity is on. This means that, for example, `LABEL` and `label` refer to different symbols. Use `-s-` to turn case sensitivity off, in which case `LABEL` and `label` will refer to the same symbol.

This option is identical to the **Case-sensitive user symbols** option in the **ASAM8** category in the IAR Embedded Workbench.

-T `-T`

This option lists active lines only.

This option is identical to the **Active lines only** option in the **ASAM8** category in the IAR Embedded Workbench.

-t `-t`*n*

By default the assembler sets 8 character positions per tab stop. The `-t` option allows you to specify a tab spacing to *n*, which must be in the range 2 to 9.

This option is useful in conjunction with the list options `-L` or `-l`; see page 15 for additional information.

This option is identical to the **Tab spacing** option in the **ASAM8** category in the IAR Embedded Workbench.

-u `-u[t|n]`

This option sets the default data pointer size to either 8 bits (tiny, `t`), or 16 bits (near, `n`).

In the IAR Embedded Workbench, this is set by the chosen data model, where `small` = `-ut`, and `large` = `-un`.

-U `-Usymbol`

Use the `-U` option to undefine the predefined symbol *`symbol`*.

By default, the assembler provides certain predefined symbols; see *Predefined symbols*, page 4. The `-U` option allows you to undefine such a predefined symbol to make its name available for your own use through a subsequent `-D` option or source definition.

Example

To use the name of the predefined symbol `__TIME__` for your own purposes, you could undefine it with:

```
asam8 prog -U __TIME__
```

-v `-v[0|1|2|3|4]`

Use the `-v` option to specify the processor configuration.

The following table shows how the `-v` options are mapped to the SAM8 derivatives:

Option	Description	Derivative
`-v0`	CPU1 type processor	SAM8 (CPU1)
`-v1`	CPU2 type processor	SAM8x (CPU2)
`-v2`	Reduced cycle count processor	SAM8xRC
`-v3`	Reduced instruction set processor	SAM8xRI
`-v4`	Reduced cycle count and instruction set processor	SAM8xRCRI

Table 12: Specifying the processor configuration (-v)

If no processor configuration option is specified, the assembler uses the `-v0` option by default.

The `-v` option is identical to the **Processor configuration** option in the **General** category in the IAR Embedded Workbench.

-w `-w[string][s]`

By default, the assembler displays a warning message when it detects an element of the source which is legal in a syntactical sense, but may contain a programming error; see *Diagnostics*, page 93, for details.

Use this option to disable warnings.

Command line option	Description
`-w`	Disables all warnings.
`-w+`	Enables all warnings.
`-w-`	Disables all warnings.
`-w+`*n*	Enables just warning *n*.
`-w-`*n*	Disables just warning *n*.
`-w+`*m*`-`*n*	Enables warnings *m* to *n*.
`-w-`*m*`-`*n*	Disables warnings *m* to *n*.

Table 13: Disabling assembler warnings (-w)

Only one `-w` option may be used on the command line.

By default, the assembler generates exit code 0 for warnings. Use the `-ws` option to generate exit code 1 if a warning message is produced.

Example

To disable just warning 0 (unreferenced label), use the following command:

```
asam8 prog -w-0
```

To disable warnings 0 to 8, use the following command:

```
asam8 prog -w-0-8
```

This option is identical to the **Warnings** option in the **ASAM8** category in the IAR Embedded Workbench.

-X

`-X`

This option includes unreferenced external symbols in the output.

-x

`-x{DI2}`

Use this option to make the assembler include a cross-reference table at the end of the list file.

This option is useful in conjunction with the list options `-L` or `-l`; see page 15 for additional information.

The following parameters are available:

Command line option	Description
`-xD`	#defines
`-xI`	Internal symbols
`-x2`	Dual line spacing

Table 14: Including cross-references in assembler list file (-x)

This option is identical to the **Include cross-reference** option in the **ASAM8** category in the IAR Embedded Workbench.

Assembler operators

This chapter first describes the precedence of the assembler operators, and then summarizes the operators, classified according to their precedence. Finally, this chapter provides reference information about each operator, presented in alphabetical order.

Precedence of operators

Each operator has a precedence number assigned to it that determines the order in which the operator and its operands are evaluated. The precedence numbers range from 1 (the highest precedence, i.e. first evaluated) to 7 (the lowest precedence, i.e. last evaluated).

The following rules determine how expressions are evaluated:

- The highest precedence operators are evaluated first, then the second highest precedence operators, and so on until the lowest precedence operators are evaluated.
- Operators of equal precedence are evaluated from left to right in the expression.
- Parentheses (and) can be used for grouping operators and operands and for controlling the order in which the expressions are evaluated. For example, the following expression evaluates to 1:

```
7/(1+(2*3))
```

Summary of assembler operators

The following tables give a summary of the operators, in order of priority. Synonyms, where available, are shown after the operator name.

UNARY OPERATORS – 1

`()`	Parenthesis.
`+`	Unary plus.
`-`	Unary minus.
`NOT, !`	Logical NOT.
`BINNOT, ~`	Bitwise NOT.
`LOW`	Low byte.
`HIGH`	High byte.

`BYTE2`	Second byte.
`BYTE3`	Third byte.
`LWRD`	Low word.
`HWRD`	High word.
`DATE`	Current time/date.
`SFB`	Segment begin.
`SFE`	Segment end.
`SIZEOF`	Segment size.

MULTIPLICATIVE AND SHIFT ARITHMETIC OPERATORS – 3

`*`	Multiplication.
`/`	Division.
`MOD, %`	Modulo.
`SHR, >>`	Logical shift right.
`SHL, <<`	Logical shift left.

ADDITIVE ARITHMETIC OPERATORS – 4

`+`	Addition.
`–`	Subtraction.

AND OPERATORS – 5

`AND, &&`	Logical AND.
`BINAND, &`	Bitwise AND.

OR OPERATORS – 6

`OR, ||`	Logical OR.
`BINOR, |`	Bitwise OR.
`XOR`	Logical exclusive OR.
`BINXOR, ^`	Bitwise exclusive OR.

COMPARISON OPERATORS – 7

`EQ, =, ==`	Equal.
`NE, <>, !=`	Not equal.
`GT, >`	Greater than.
`LT, <`	Less than.
`UGT`	Unsigned greater than.
`ULT`	Unsigned less than.
`GE, >=`	Greater than or equal.
`LE, <=`	Less than or equal.

Description of operators

The following sections give detailed descriptions of each assembler operator. See *Assembler expressions*, page 2, for related information.

`()` Parenthesis (1).

`(` and `)` group expressions to be evaluated separately, overriding the default precedence order.

Example

```
1+2*3 → 7
(1+2)*3 → 9
```

`*` Multiplication (3).

`*` produces the product of its two operands. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

Example

```
2*2 → 4
-2*2 → -4
```

`+` Unary plus (1).

Unary plus operator.

Example

```
+3 → 3
3*+2 → 6
```

\+ Addition (4).

The + addition operator produces the sum of the two operands which surround it. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

Example

```
92+19 → 111
-2+2 → 0
-2+-2 → -4
```

\- Unary minus (1).

The unary minus operator performs arithmetic negation on its operand.

The operand is interpreted as a 32-bit signed integer and the result of the operator is the two's complement negation of that integer.

\- Subtraction (4).

The subtraction operator produces the difference when the right operand is taken away from the left operand. The operands are taken as signed 32-bit integers and the result is also signed 32-bit integer.

Example

```
92-19 → 73
-2-2 → -4
-2--2 → 0
```

/ Division (3).

/ produces the integer quotient of the left operand divided by the right operator. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

Example

```
9/2 → 4
-12/3 → -4
9/2*6 → 24
```

`<, LT` Less than (7).

`<` evaluates to 1 (true) if the left operand has a lower numeric value than the right operand.

Example

```
-1 < 2 → 1
2 < 1 → 0
2 < 2 → 0
```

`<=, LE` Less than or equal (7).

`<=` evaluates to 1 (true) if the left operand has a lower or equal numeric value to the right operand.

Example

```
1 <= 2 → 1
2 <= 1 → 0
1 <= 1 → 1
```

`<>, !=, NE` Not equal (7).

`<>` evaluates to 0 (false) if its two operands are identical in value or to 1 (true) if its two operands are not identical in value.

Example

```
1 <> 2 → 1
2 <> 2 → 0
'A' <> 'B' → 1
```

`=, ==, EQ` Equal (7).

`=` evaluates to 1 (true) if its two operands are identical in value, or to 0 (false) if its two operands are not identical in value.

Example

```
1 = 2 → 0
2 == 2 → 1
'ABC' = 'ABCD' → 0
```

`>`, `GT` Greater than (7).

`>` evaluates to 1 (true) if the left operand has a higher numeric value than the right operand.

Example

```
-1 > 1 → 0
2 > 1 → 1
1 > 1 → 0
```

`>=`, `GE` Greater than or equal (7).

`>=` evaluates to 1 (true) if the left operand is equal to or has a higher numeric value than the right operand.

Example

```
1 >= 2 → 0
2 >= 1 → 1
1 >= 1 → 1
```

`&&`, `AND` Logical AND (5).

Use `&&` to perform logical AND between its two integer operands. If both operands are non-zero the result is 1; otherwise it is zero.

Example

```
B'1010 && B'0011 → 1
B'1010 && B'0101 → 1
B'1010 && B'0000 → 0
```

`&`, `BINAND` Bitwise AND (5).

Use `&` to perform bitwise AND between the integer operands.

Example

```
B'1010 & B'0011 → B'0010
B'1010 & B'0101 → B'0000
B'1010 & B'0000 → B'0000
```

`~, BINNOT` Bitwise NOT (1).

Use `~` to perform bitwise NOT on its operand.

Example

```
~ B'1010 → B'11111111111111111111111111110101
```

`|, BINOR` Bitwise OR (6).

Use `|` to perform bitwise OR on its operands.

Example

```
B'1010 | B'0101 → B'1111
B'1010 | B'0000 → B'1010
```

`^, BINXOR` Bitwise exclusive OR (6).

Use `^` to perform bitwise XOR on its operands.

Example

```
B'1010 ^ B'0101 → B'1111
B'1010 ^ B'0011 → B'1001
```

`%, MOD` Modulo (3).

`%` produces the remainder from the integer division of the left operand by the right operand. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

`X % Y` is equivalent to `X-Y*(X/Y)` using integer division.

Example

```
2 % 2 → 0
12 % 7 → 5
3 % 2 → 1
```

`!, NOT` Logical NOT (1).

Use `!` to negate a logical argument.

Example

```
! B'0101 → 0
! B'0000 → 1
```

`||`, `OR` Logical OR (6).

Use `||` to perform a logical OR between two integer operands.

Example

```
B'1010 || B'0000 → 1
B'0000 || B'0000 → 0
```

`BYTE2` Second byte (1).

`BYTE2` takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the middle-low byte (bits 15 to 8) of the operand.

Example

```
BYTE2 0x12345678 → 0x56
```

`BYTE3` Third byte (1).

`BYTE3` takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the middle-high byte (bits 23 to 16) of the operand.

Example

```
BYTE3 0x12345678 → 0x34
```

`DATE` Current time/date (1).

Use the `DATE` operator to specify when the current assembly began.

The `DATE` operator takes an absolute argument (expression) and returns:

`DATE 1`	Current second (0–59).
`DATE 2`	Current minute (0–59).
`DATE 3`	Current hour (0–23).
`DATE 4`	Current day (1–31).

`DATE 5`	Current month (1–12).
`DATE 6`	Current year MOD 100 (1998 → 98, 2000 → 00, 2002 → 02).

Example

To assemble the date of assembly:

```
today: DC8 DATE 6, DATE 5, DATE 4
```

`HIGH` High byte (1).

`HIGH` takes a single operand to its right which is interpreted as an unsigned, 16-bit integer value. The result is the unsigned 8-bit integer value of the higher order byte of the operand.

Example

`HIGH 0xABCD` → `0xAB`

`HWRD` High word (1).

`HWRD` takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the high word (bits 31 to 16) of the operand.

Example

`HWRD 0x12345678` → `0x1234`

`LOW` Low byte (1).

`LOW` takes a single operand, which is interpreted as an unsigned, 16-bit integer value. The result is the unsigned, 8-bit integer value of the lower order byte of the operand.

Example

`LOW 0xABCD` → `0xCD`

`LWRD` Low word (1).

`LWRD` takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the low word (bits 15 to 0) of the operand.

Example

`LWRD 0x12345678` → `0x5678`

`SFB` Segment begin (1).

Syntax

SFB(*segment* [{+ | -} *offset*])

Parameters

segment	The name of a relocatable segment, which must be defined before `SFB` is used.
offset	An optional offset from the start address. The parentheses are optional if *offset* is omitted.

Description

`SFB` accepts a single operand to its right. The operand must be the name of a relocatable segment.

The operator evaluates to the absolute address of the first byte of that segment. This evaluation takes place at linking time.

Example

```
       NAME   demo
       RSEG   CODE
start: DC16   SFB(CODE)
```

Even if the above code is linked with many other modules, `start` will still be set to the address of the first byte of the segment.

`SFE` Segment end (1).

Syntax

SFE (*segment* [{+ | -} *offset*])

Parameters

segment	The name of a relocatable segment, which must be defined before `SFE` is used.

offset An optional offset from the start address. The parentheses are optional if `offset` is omitted.

Description

`SFE` accepts a single operand to its right. The operand must be the name of a relocatable segment. The operator evaluates to the segment start address plus the segment size. This evaluation takes place at linking time.

Example

```
      NAME   demo
      RSEG   CODE
end:  DC16   SFE(CODE)
```

Even if the above code is linked with many other modules, end will still be set to the address of the last byte of the segment.

The size of the segment `MY_SEGMENT` can be calculated as:

```
SFE(MY_SEGMENT)-SFB(MY_SEGMENT)
```

`<<`, `SHL` Logical shift left (3).

Use `<<` to shift the left operand, which is always treated as `unsigned`, to the left. The number of bits to shift is specified by the right operand, interpreted as an integer value between 0 and 32.

Example

```
B'00011100 << 3 → B'11100000
B'0000011111111111 << 5 → B'1111111111100000
14 << 1 → 28
```

`>>`, `SHR` Logical shift right (3).

Use `>>` to shift the left operand, which is always treated as `unsigned`, to the right. The number of bits to shift is specified by the right operand, interpreted as an integer value between 0 and 32.

Example

```
B'01110000 >> 3 → B'00001110
B'1111111111111111 >> 20 → 0
14 >> 1 → 7
```

SIZEOF Segment size (1).

Syntax

`SIZEOF segment`

Parameters

segment	The name of a relocatable segment, which must be defined before `SIZEOF` is used.

Description

`SIZEOF` generates `SFE-SFB` for its argument, which should be the name of a relocatable segment; i.e. it calculates the size in bytes of a segment. This is done when modules are linked together.

Example

```
      NAME   demo
      RSEG   CODE
size: DC16   SIZEOF CODE
```

sets `size` to the size of segment `CODE`.

UGT Unsigned greater than (7).

`UGT` evaluates to 1 (true) if the left operand has a larger value than the right operand. The operation treats its operands as unsigned values.

Example

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

ULT Unsigned less than (7).

`ULT` evaluates to 1 (true) if the left operand has a smaller value than the right operand. The operation treats its operands as unsigned values.

Example

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

XOR Logical exclusive OR (6).

Use XOR to perform logical XOR on its two operands.

Example

```
B'0101 XOR B'1010 → 0
B'0101 XOR B'0000 → 1
```

Assembler directives

This chapter gives an alphabetical summary of the assembler directives. It then describes the syntax conventions and provides detailed reference information for each category of directives.

Summary of assembler directives

The following table gives a summary of all the assembler directives.

Directive	Description	Section
`_args`	Is set to number of arguments passed to macro.	Macro processing
`$`	Includes a file.	Assembler control
`#define`	Assigns a value to a label.	C-style preprocessor
`#elif`	Introduces a new condition in a `#if`...`#endif` block.	C-style preprocessor
`#else`	Assembles instructions if a condition is false.	C-style preprocessor
`#endif`	Ends a `#if`, `#ifdef`, or `#ifndef` block.	C-style preprocessor
`#error`	Generates an error.	C-style preprocessor
`#if`	Assembles instructions if a condition is true.	C-style preprocessor
`#ifdef`	Assembles instructions if a symbol is defined.	C-style preprocessor
`#ifndef`	Assembles instructions if a symbol is undefined.	C-style preprocessor
`#include`	Includes a file.	C-style preprocessor
`#pragma`	Ignored.	C-style preprocessor
`#undef`	Undefines a label.	C-style preprocessor
`/*comment*/`	C-style comment delimiter.	C-style preprocessor
`//`	C++ style comment delimiter.	C-style preprocessor
`=`	Assigns a permanent value local to a module.	Value assignment
`ALIAS`	Assigns a permanent value local to a module.	Value assignment
`ALIGN`	Aligns the location counter by inserting zero-filled bytes.	Segment control
`ALIGNRAM`	Aligns the location counter by incrementing it (no filling).	Segment control
`ARGFRAME`	Defines a function's arguments.	Function control

Table 15: Assembler directives summary

Directive	Description	Section
ASEG	Begins an absolute segment.	Segment control
ASEGN	Begins a named absolute segment.	Segment control
ASSIGN	Assigns a temporary value.	Value assignment
CASEOFF	Disables case sensitivity.	Assembler control
CASEON	Enables case sensitivity.	Assembler control
CFI	Specifies call frame information.	Call frame information
COL	Sets the number of columns per page.	Listing control
COMMON	Begins a common segment.	Segment control
CONST	Specifies an SFR label as read-only.	Value assignment
DB	Generates 8-bit byte constants, including strings.	Space allocation
DC16	Generates 16-bit constants.	Space allocation
DC24	Generates 24-bit constants.	Space allocation
DC32	Generates 32-bit constants.	Space allocation
DC8	Generates 8-bit byte constants, including strings.	Space allocation
DD	Generates 32-bit constants.	Space allocation
DECLARE	Defines a file-wide value with optional r or R prefix.	Value assignment
DEFINE	Defines a file-wide value.	Value assignment
DP	Generates 24-bit constants.	Space allocation
DS	Reserves memory space without initializing (8-bit).	Space allocation
DS8	Reserves memory space without initializing (8-bit).	Space allocation
DW	Generates 16-bit constants.	Space allocation
ELSE	Assembles instructions if a condition is false.	Conditional assembly
ELSEIF	Specifies a new condition in an IF...ENDIF block.	Conditional assembly
END	Terminates the assembly of the last module in a file.	Module control
ENDIF	Ends an IF block.	Conditional assembly
ENDM	Ends a macro definition.	Macro processing
ENDMOD	Terminates the assembly of the current module.	Module control

Table 15: Assembler directives summary (Continued)

Directive	Description	Section
ENDR	Ends a repeat structure.	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment
EVEN	Aligns the program counter to an even address.	Segment control
EXITM	Exits prematurely from a macro.	Macro processing
EXPORT	Exports symbols to other modules.	Symbol control
EXTERN	Imports an external symbol.	Symbol control
FUNCALL	Defines function call information.	Function control
FUNCTION	Defines a function.	Function control
IF	Assembles instructions if a condition is true.	Conditional assembly
IMPORT	Imports an external symbol.	Symbol control
LIBRARY	Begins a library module.	Module control
LIMIT	Checks a value against limits.	Value assignment
LOCAL	Creates symbols local to a macro.	Macro processing
LOCFRAME	Defines a function's local variables.	Function control
LSTCND	Controls conditional assembler listing.	Listing control
LSTCOD	Controls multi-line code listing.	Listing control
LSTEXP	Controls the listing of macro generated lines.	Listing control
LSTMAC	Controls the listing of macro definitions.	Listing control
LSTOUT	Controls assembler-listing output.	Listing control
LSTPAG	Controls the formatting of output into pages.	Listing control
LSTREP	Controls the listing of lines generated by repeat directives.	Listing control
LSTXRF	Generates a cross-reference table.	Listing control
MACRO	Defines a macro.	Macro processing
MODULE	Begins a library module.	Module control
NAME	Begins a program module.	Module control
ORG	Sets the location counter.	Segment control
PAGE	Generates a new page.	Listing control
PAGSIZ	Sets the number of lines per page.	Listing control
PROGRAM	Begins a program module.	Module control
PUBLIC	Exports symbols to other modules.	Symbol control

Table 15: Assembler directives summary (Continued)

Directive	Description	Section
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.	Symbol control
RADIX	Sets the default base.	Assembler control
REPT	Assembles instructions a specified number of times.	Macro processing
REPTC	Repeats and substitutes characters.	Macro processing
REPTI	Repeats and substitutes strings.	Macro processing
REQUIRE	Forces a symbol to be referenced.	Symbol control
RSEG	Begins a relocatable segment.	Segment control
RTMODEL	Declares runtime model attributes.	Module control
SET	Assigns a temporary value.	Value assignment
SFR	Creates byte-access SFR labels.	Value assignment
SFRP	Creates word-access SFR labels.	Value assignment
SFRTYPE	Specifies SFR attributes.	Value assignment
STACK	Begins a stack segment.	Segment control
VAR	Assigns a temporary value.	Value assignment

Table 15: Assembler directives summary (Continued)

Syntax conventions

In the syntax definitions the following conventions are used:

- Parameters, representing what you would type, are shown in italics. So, for example, in:
  `ORG` *`expr`*
  `expr` represents an arbitrary expression.
- Optional parameters are shown in square brackets. So, for example, in:
  `END [`*`expr`*`]`

  the *`expr`* parameter is optional. An ellipsis indicates that the previous item can be repeated an arbitrary number of times. For example:

  `PUBLIC` *`symbol`* `[,`*`symbol`*`] …`

  indicates that `PUBLIC` can be followed by one or more symbols, separated by commas.

- Alternatives are enclosed in { and } brackets, separated by a vertical bar, for example:
  `LSTOUT{+|-}`
  indicates that the directive must be followed by either `+` or `-`.

LABELS AND COMMENTS

Where a label *must* precede a directive, this is indicated in the syntax, as in:

`label` `VAR` *`expr`*

An optional label, which will assume the value and type of the current program location counter (`PLC`), can precede all directives. For clarity, this is not included in each syntax definition.

In addition, unless explicitly specified, all directives can be followed by a comment, preceded by `;` (semicolon).

PARAMETERS

The following table shows the correct form of the most commonly used types of parameter:

Parameter	What it consists of
`expr`	An expression; see *Assembler expressions*, page 2.
`label`	A symbolic label.
`symbol`	An assembler symbol.

Table 16: Assembler directive parameters

Module control directives

Module control directives are used for marking the beginning and end of source program modules, and for assigning names and types to them.

Directive	Description
`END`	Terminates the assembly of the last module in a file.
`ENDMOD`	Terminates the assembly of the current module.
`LIBRARY`	Begins a library module.
`MODULE`	Begins a library module.
`NAME`	Begins a program module.
`PROGRAM`	Begins a program module.
`RTMODEL`	Declares runtime model attributes.

Table 17: Module control directives

SYNTAX

```
END [label]
ENDMOD [label]
LIBRARY symbol [(expr)]
MODULE symbol [(expr)]
NAME symbol [(expr)]
PROGRAM symbol [(expr)]
RTMODEL key, value
```

PARAMETERS

expr	Optional expression (0–255) used by the IAR compiler to encode programming language, memory model, and processor configuration.
key	A text string specifying the key.
label	An expression or label that can be resolved at assembly time. It is output in the object code as a program entry address.
symbol	Name assigned to module, used by XLINK and XLIB when processing object files.
value	A text string specifying the value.

DESCRIPTION

Beginning a program module

Use `NAME` to begin a program module, and to assign a name for future reference by the IAR XLINK Linker™ and the IAR XLIB Librarian™.

Program modules are unconditionally linked by XLINK, even if other modules do not reference them.

Beginning a library module

Use `MODULE` to create libraries containing lots of small modules—like runtime systems for high-level languages—where each module often represents a single routine. With the multi-module facility, you can significantly reduce the number of source and object files needed.

Library modules are only copied into the linked code if other modules reference a public symbol in the module.

Terminating a module

Use `ENDMOD` to define the end of a module.

Terminating the last module

Use END to indicate the end of the source file. Any lines after the END directive are ignored.

Assembling multi-module files

Program entries must be either relocatable or absolute, and will show up in XLINK load maps, as well as in some of the hexadecimal absolute output formats. Program entries must not be defined externally.

The following rules apply when assembling multi-module files:

- At the beginning of a new module all user symbols are deleted, except for those created by DEFINE, #define, or MACRO, the location counters are cleared, and the mode is set to absolute.
- Listing control directives remain in effect throughout the assembly.

Note: END must always be used in the *last* module, and there must not be any source lines (except for comments and listing control directives) between an ENDMOD and a MODULE directive.

If the NAME or MODULE directive is missing, the module will be assigned the name of the source file and the attribute program.

Declaring runtime model attributes

Use RTMODEL to enforce consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key value, or the special value *. Using the special value * is equivalent to not defining the attribute at all. It can however be useful to explicitly state that the module can handle any runtime model.

A module can have several runtime model definitions.

Note: The compiler runtime model attributes start with double underscore. In order to avoid confusion, this style must not be used in the user-defined assembler attributes.

If you are writing assembler routines for use with C code, and you want to control the module consistency, refer to the *SAM8 IAR C Compiler Reference Guide.*

Examples

The following example defines three modules where:

- MOD_1 and MOD_2 *cannot* be linked together since they have different values for runtime model "foo".
- MOD_1 and MOD_3 *can* be linked together since they have the same definition of runtime model "bar" and no conflict in the definition of "foo".

- `MOD_2` and `MOD_3` *can* be linked together since they have no runtime model conflicts. The value "*" matches any runtime model value.

```
MODULE MOD_1
  RTMODEL   "foo", "1"
  RTMODEL   "bar", "XXX"
  ...
ENDMOD

MODULE MOD_2
  RTMODEL   "foo", "2"
  RTMODEL   "bar", "*"
  ...
ENDMOD

MODULE MOD_3
  RTMODEL   "bar", "XXX"
  ...
END
```

Symbol control directives

These directives control how symbols are shared between modules.

Directive	Description
`EXTERN (IMPORT)`	Imports an external symbol.
`PUBLIC (EXPORT)`	Exports symbols to other modules.
`PUBWEAK`	Exports symbols to other modules, multiple definitions allowed.
`REQUIRE`	Forces a symbol to be referenced.

Table 18: Symbol control directives

SYNTAX

```
EXTERN symbol [,symbol] …
PUBLIC symbol [,symbol] …
PUBWEAK symbol [,symbol] …
REQUIRE symbol
```

PARAMETERS

symbol	Symbol to be imported or exported.

DESCRIPTION

Exporting symbols to other modules

Use PUBLIC to make one or more symbols available to other modules. The symbols declared as PUBLIC can only be assigned values by using them as labels. Symbols declared PUBLIC can be relocated or absolute, and can also be used in expressions (with the same rules as for other symbols).

The PUBLIC directive always exports full 32-bit values, which makes it feasible to use global 32-bit constants also in assemblers for 8-bit and 16-bit processors. With the LOW, HIGH, >>, and << operators, any part of such a constant can be loaded in an 8-bit or 16-bit register or word.

There are no restrictions on the number of PUBLIC-declared symbols in a module.

Importing symbols

Use EXTERN to import an untyped external symbol.

EXAMPLES

The following example defines a subroutine to print an error message, and exports the entry address err so that it can be called from other modules. It defines print as an external routine; the address will be resolved at link time.

```
     NAME   error
     EXTERN print
     PUBLIC err

err  RCALL  print
     DC8    "** Error **"
     EVEN
     RET

     END
```

Segment control directives

The segment directives control how code and data are generated.

Directive	Description
ALIGN	Aligns the location counter by inserting zero-filled bytes.
ALIGNRAM	Aligns the location counter by incrementing it (no filling).
ASEG	Begins an absolute segment.

Table 19: Segment control directives

Directive	Description
ASEGN	Begins a named absolute segment.
COMMON	Begins a common segment.
EVEN	Aligns the program counter to an even address.
ORG	Sets the location counter.
RSEG	Begins a relocatable segment.
STACK	Begins a stack segment.

Table 19: Segment control directives (Continued)

SYNTAX

```
ALIGN align [,value]
ALIGNRAM align [,value]
ASEG [start [(align)]]
ASEGN segment [:type], address
COMMON segment [:type] [(align)]
EVEN [value]
ORG expr
RSEG segment [:type] [flag] [(align)]
RSEG segment [:type], address
STACK segment [:type] [(align)]
```

PARAMETERS

address	Address where this segment part will be placed.
align	Exponent of the value to which the address should be aligned, in the range 0 to 30. For example, `align 1` results in word alignment `2`.
expr	Address to set the location counter to.

flag	`NOROOT`
	This segment part may be discarded by the linker even if no symbols in this segment part are referred to. Normally 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. For a given segment, all segment parts must specify the same state for this flag. The default mode is `NOREORDER` which indicates that the segment parts must remain in order.

	`SORT` The linker will sort the segment parts in decreasing alignment order. For a given segment, all segment parts must specify the same state for this flag. The default mode is `NOSORT` which indicates that the segment parts will not be sorted.
`segment`	The name of the segment.
`start`	A start address that has the same effect as using an `ORG` directive at the beginning of the absolute segment.
`type`	The memory type, typically `CODE`, or `DATA`. In addition, any of the types supported by the IAR XLINK Linker.
`value`	Byte value used for padding, default is zero.

DESCRIPTION

Beginning an absolute segment

Use `ASEG` to set the absolute mode of assembly, which is the default at the beginning of a module.

If the parameter is omitted, the start address of the first segment is 0, and subsequent segments continue after the last address of the previous segment.

Beginning a named absolute segment

Use `ASEGN` to start a named absolute segment located at the address *`address`*. This directive has the advantage of allowing you to specify the memory type of the segment.

Beginning a relocatable segment

Use `RSEG` to set the current mode of the assembly to relocatable assembly mode. The assembler maintains separate location counters (initially set to zero) for all segments, which makes it possible to switch segments and mode anytime without the need to save the current segment location counter.

Up to 65536 unique, relocatable segments may be defined in a single module.

Beginning a stack segment

Use `STACK` to allocate code or data allocated from high to low addresses (in contrast with the `RSEG` directive that causes low-to-high allocation).

Note: The contents of the segment are not generated in reverse order.

Beginning a common segment

Use `COMMON` to place data in memory at the same location as `COMMON` segments from other modules that have the same name. In other words, all `COMMON` segments of the same name will start at the same location in memory and overlay each other.

Obviously, the `COMMON` segment type should not be used for overlaid executable code. A typical application would be when you want a number of different routines to share a reusable, common area of memory for data.

It can be practical to have the interrupt vector table in a `COMMON` segment, thereby allowing access from several routines.

The final size of the `COMMON` segment is determined by the size of largest occurrence of this segment. The location in memory is determined by the XLINK `-Z` command; see the *IAR Linker and Library Tools Reference Guide*.

Use the *`align`* parameter in any of the above directives to align the segment start address.

Setting the program location counter (PLC)

Use `ORG` to set the program location counter of the current segment to the value of an expression. The optional label will assume the value and type of the new location counter.

The result of the expression must be of the same type as the current segment, i.e. it is not valid to use `ORG 10` during `RSEG`, since the expression is absolute; use `ORG $+10` instead. The expression must not contain any forward or external references.

All program location counters are set to zero at the beginning of an assembly module.

Aligning a segment

Use `ALIGN` to align the program location counter to a specified address boundary. The expression gives the power of two to which the program counter should be aligned.

The alignment is made relative to the segment start; normally this means that the segment alignment must be at least as large as that of the alignment directive to give the desired result.

`ALIGN` aligns by inserting zero/filled bytes. The `EVEN` directive aligns the program counter to an even address (which is equivalent to `ALIGN 1`) and the `ODD` directive aligns the program counter to an odd address.

EXAMPLES

Beginning an absolute segment

The following example assembles interrupt routine entry instructions in the appropriate interrupt vectors using an absolute segment:

```
      EXTERN   reset,IRQ0,IRQ1,IRQ2

      ASEG
      ORG      0x00
int0  DC16     IRQ0
int1  DC16     IRQ1
int2  DC16     IRQ2
      ;...etc

      ORG      0x100
      JP       T,reset    ; Reset vector

      END
```

Beginning a relocatable segment

In the following example, the data following the first `RSEG` directive is placed in a relocatable segment called `table`; the `ORG` directive is used for creating a gap of six bytes in the table.

The code following the second `RSEG` directive is placed in a relocatable segment called `code`:

```
        EXTERN   divrtn,mulrtn

        RSEG     table
        DC16     divrtn,mulrtn
        ORG      $+6
        DC16     subrtn

        RSEG     code
subrtn  MOV      R6,R7
        SUBI     R6,#20
        END
```

Beginning a stack segment

The following example defines two 100-byte stacks in a relocatable segment called `rpnstack`:

```
         STACK   rpnstack
parms    DS8     100
opers    DS8     100
         END
```

The data is allocated from high to low addresses.

Beginning a common segment

The following example defines two common segments containing variables:

```
         NAME    common1
         COMMON  data
count    DD      1
         ENDMOD

         NAME    common2
         COMMON  data
up       DC8     1
         ORG     $+2
down     DC8     1
         END
```

Because the common segments have the same name, `data`, the variables `up` and `down` refer to the same locations in memory as the first and last bytes of the 4-byte variable `count`.

Aligning a segment

This example starts a relocatable segment, moves to an even address, and adds some data. It then aligns to a 64-byte boundary before creating a 64-byte table.

```
         NAME     align
         RSEG     data     ; Start a relocatable data segment

         EVEN              ; Ensure it's on an even boundary
target   DC16     1        ; Target is on an even boundary

         ALIGN    6        ; Zero-fill to a 64-byte boundary
results  DS8      64       ; Create a 64-byte table

         ALIGNRAM 3        ; Align to an 8-byte boundary
ages     DS8      64       ; Create another 64-byte table
         END
```

Value assignment directives

These directives are used for assigning values to symbols.

Directive	Description
`=`	Assigns a permanent value local to a module.
`ALIAS`	Assigns a permanent value local to a module.
`ASSIGN`	Assigns a temporary value.
`CONST`	Specifies an SFR label as read-only.
`DECLARE`	Defines a file-wide value with optional `r` or `R` prefix.
`DEFINE`	Defines a file-wide value.
`EQU`	Assigns a permanent value local to a module.
`LIMIT`	Checks a value against limits.
`SET`	Assigns a temporary value.
`SFR`	Creates byte-access SFR labels.
`SFRTYPE`	Specifies SFR attributes.
`SFRP`	Creates word-access SFR labels.
`VAR`	Assigns a temporary value.

Table 20: Value assignment directives

SYNTAX

```
label = expr
label ALIAS expr
label ASSIGN expr
label CONST expr
DECLARE label, expr
label DEFINE expr
label EQU expr
LIMIT expr, min, max, message
label SET expr
[const] SFR register = value
[const] SFRTYPE register attribute [,attribute] = value
[const] SFRP register = value
label VAR expr
```

PARAMETERS

attribute	One or more of the following:	
	BYTE	The SFR must be accessed as a byte.
	READ	You can read from this SFR.
	WORD	The SFR must be accessed as a word.
	WRITE	You can write to this SFR.
expr	Value assigned to symbol or value to be tested.	
label	Symbol to be defined.	
message	A text message that will be printed when *expr* is out of range.	
min, max	The minimum and maximum values allowed for *expr*.	
register	The special function register.	
value	The SFR port address.	

DESCRIPTION

Defining a temporary value

Use either of ASSIGN, SET, and VAR to define a symbol that may be redefined, such as for use with macro variables. Symbols defined with VAR cannot be declared PUBLIC.

Defining a permanent local value

Use EQU or = to assign a value to a symbol.

Use EQU to create a local symbol that denotes a number or offset.

The symbol is only valid in the module in which it was defined, but can be made available to other modules with a PUBLIC directive.

Use EXTERN to import symbols from other modules.

Defining a permanent global value

Use DEFINE or DECLARE to define symbols that should be known to all modules in the source file. Symbols defined with DECLARE can optionally be prefixed with r or R.

A symbol which has been given a value with DEFINE or DECLARE can be made available to modules in other files with the PUBLIC directive.

Symbols defined with DEFINE or DECLARE cannot be redefined.

Defining special function registers

Use SFR to create special function register labels with attributes READ, WRITE, and BYTE turned on. Use SFRP to create special function register labels with attributes READ, WRITE, or WORD turned on. Use SFRTYPE to create special function register labels with specified attributes.

Prefix the directive with CONST to disable the WRITE attribute assigned to the SFR. You will then get an error/warning when trying to write to the SFR.

Checking symbol values

Use LIMIT to check that expressions lie within a specified range. If the expression is assigned a value outside the range, an error message will appear.

The check will occur as soon as the expression is resolved, which will be during linking if the expression contains external references. The *min* and *max* expressions cannot involve references to forward or external labels, i.e. they must be resolved when encountered.

EXAMPLES

Defining a permanent global value

```
globvalue  DEFINE  12
DECLARE    REG4,   4
```

Redefining a symbol

The following example uses VAR to redefine the symbol cons in a REPT loop to generate a table of the first 8 powers of 3:

```
         NAME     table
cons     VAR      1
buildit  MACRO    times
         DC16     cons
cons     VAR      cons*3
         IF       times>1
         buildit  times-1
         ENDIF
         ENDM
main     buildit 4
         END
```

It generates the following code:

```
  1    00000000                     NAME    table
  2    00000001              cons   VAR     1
 10    00000000              main   buildit 4
 10.1  00000000 0001                DC16    cons
 10.2  00000003              cons   VAR     cons*3
 10.3  00000002                     IF      4>1
 10    00000002                     buildit 4-1
 10.1  00000002 0003                DC16    cons
 10.2  00000009              cons   VAR     cons*3
 10.3  00000004                     IF      4-1>1
 10    00000004                     buildit 4-1-1
 10.1  00000004 0009                DC16    cons
 10.2  0000001B              cons   VAR     cons*3
 10.3  00000006                     IF      4-1-1>1
 10    00000006                     buildit 4-1-1-1
 10.1  00000006 001B                DC16    cons
 10.2  00000051              cons   VAR     cons*3
 10.3  00000008                     IF      4-1-1-1>1
 10.4  00000008                     buildit 4-1-1-1-1
 10.5  00000008                     ENDIF
 10.6  00000008                     ENDM
 10.7  00000008                     ENDIF
 10.8  00000008                     ENDM
 10.9  00000008                     ENDIF
 10.10 00000008                     ENDM
 10.11 00000008                     ENDIF
 10.12 00000008                     ENDM
 11    00000008                     END
```

Using local and global symbols

In the following example the symbol `value` defined in module `add1` is local to that module; a distinct symbol of the same name is defined in module `add2`. The `DEFINE` directive is used for declaring `locn` for use anywhere in the file:

```
         NAME   add1
locn     DEFINE 020h
value    EQU    77
         CLR    R10
         LD     R11, #locn
         LDC    R6,@RR10
         LD     R7, #value
         ADD    R6,R7
         RET
         ENDMOD
```

```
        NAME    add2
value   EQU     88
        CLR     R10
        LD      R11, #locn
        LDC     R6, @RR10
        LD      R7, #value
        ADD     R6,R7
        RET
        END
```

The symbol `locn` defined in module `add1` is also available to module `add2`.

Using special function registers

In this example a number of SFR variables are declared with a variety of access capabilities:

```
sfrb portd                = 0x12       /* byte read/write
                                                  access */
sfrw ocr1                 = 0x2A       /* word read/write
                                                  access */
const sfrb pind           = 0x10       /* byte read only
                                              access */
SFRTYPE portb write, byte = 0x18       /* byte write only
                                          access */
```

Using the LIMIT directive

The following example sets the value of a variable called `speed` and then checks it, at assembly time, to see if it is in the range 10 to 30. This might be useful if `speed` is often changed at compile time, but values outside a defined range would cause undesirable behavior.

```
speed     VAR       23
LIMIT     speed,10,30,...speed out of range...
```

Conditional assembly directives

These directives provide logical control over the selective assembly of source code.

Directive	Description
`IF`	Assembles instructions if a condition is true.
`ELSE`	Assembles instructions if a condition is false.
`ELSEIF`	Specifies a new condition in an `IF`…`ENDIF` block.
`ENDIF`	Ends an `IF` block.

Table 21: Conditional assembly directives

SYNTAX

```
IF condition
ELSE
ELSEIF condition
ENDIF
```

PARAMETERS

condition	One of the following:	
	An absolute expression	The expression must not contain forward or external references, and any non-zero value is considered as true.
	string1=string2	The condition is true if *string1* and *string2* have the same length and contents.
	string1<>string2	The condition is true if *string1* and *string2* have different length or contents.

DESCRIPTION

Use the `IF`, `ELSE`, and `ENDIF` directives to control the assembly process at assembly time. If the condition following the `IF` directive is not true, the subsequent instructions will not generate any code (i.e. it will not be assembled or syntax checked) until an `ELSE` or `ENDIF` directive is found.

Use `ELSEIF` to introduce a new condition after an `IF` directive. Conditional assembler directives may be used anywhere in an assembly, but have their greatest use in conjunction with macro processing.

All assembler directives (except `END`) as well as the inclusion of files may be disabled by the conditional directives. Each `IF` directive must be terminated by an `ENDIF` directive. The `ELSE` directive is optional, and if used, it must be inside an `IF...ENDIF` block. `IF...ENDIF` and `IF...ELSE...ENDIF` blocks may be nested to any level.

EXAMPLES

The following macro subtracts a constant from any byte register.

```
sub  MACRO   r, c
     IF      c=1
     DEC     r
     ELSE
     SUB     r, #c
```

```
     ENDIF
     ENDM
```

If the argument to the macro is `1` it generates a `DEC` instruction to save instruction cycles; otherwise it generates a `SUB` intruction.

It could be tested with the following program:

```
main LD      R6, #17
     sub     R6, 2
     LD      R7, #22
     sub     R7, 1
     RET

     END
```

Macro processing directives

These directives allow user macros to be defined.

Directive	Description
`_args`	Is set to number of arguments passed to macro.
`ENDM`	Ends a macro definition.
`ENDR`	Ends a repeat structure.
`EXITM`	Exits prematurely from a macro.
`LOCAL`	Creates symbols local to a macro.
`MACRO`	Defines a macro.
`REPT`	Assembles instructions a specified number of times.
`REPTC`	Repeats and substitutes characters.
`REPTI`	Repeats and substitutes strings.

Table 22: Macro processing directives

SYNTAX

```
ENDM
ENDR
EXITM
LOCAL symbol [,symbol] …
name MACRO [,argument] …
REPT expr
REPTC formal,actual
REPTI formal,actual [,actual] …
```

PARAMETERS

actual	String to be substituted.
argument	A symbolic argument name.
expr	An expression.
formal	Argument into which each character of *actual* (REPTC) or each *actual* (REPTI) is substituted.
name	The name of the macro.
symbol	Symbol to be local to the macro.

DESCRIPTION

A macro is a user-defined symbol that represents a block of one or more assembler source lines. Once you have defined a macro you can use it in your program like an assembler directive or assembler mnemonic.

When the assembler encounters a macro, it looks up the macro's definition, and inserts the lines that the macro represents as if they were included in the source file at that position.

Macros perform simple text substitution effectively, and you can control what they substitute by supplying parameters to them.

Defining a macro

You define a macro with the statement:

macroname MACRO [,*arg*] [,*arg*] ...

Here *macroname* is the name you are going to use for the macro, and *arg* is an argument for values that you want to pass to the macro when it is expanded.

Insert the target-specific file macro.fm here:

For example, you could define a macro ERROR as follows:

```
errmac  MACRO   text
        CALL    abort
        DC8     text,0
        ENDM
```

This macro uses a parameter text to set up an error message for a routine abort. You would call the macro with a statement such as:

```
        errmac  'Disk not ready'
```

The assembler will expand this to:

```
        CALL    abort
        DC8     'Disk not ready',0
```

If you omit a list of one or more arguments, the arguments you supply when calling the macro are called `\0` to `\9` and `\A` to `\Z`.

The previous example could therefore be written as follows:

```
errmac  MACRO
        CALL    abort
        DC8     \0,0
        ENDM
```

Use the `EXITM` directive to generate a premature exit from a macro.

`EXITM` is not allowed inside `REPT`...`ENDR`, `REPTC`...`ENDR`, or `REPTI`...`ENDR` blocks.

Use `LOCAL` to create symbols local to a macro. The `LOCAL` directive must be used before the symbol is used.

Each time that a macro is expanded, new instances of local symbols are created by the `LOCAL` directive. Therefore, it is legal to use local symbols in recursive macros.

Note: It is illegal to *redefine* a macro.

Passing special characters

Macro arguments that include commas or white space can be forced to be interpreted as one argument by using the matching quote characters `<` and `>` in the macro call.

Import the target-specific file macroqch.fm here:

For example:

```
macld    MACRO  op
         LD     op
         ENDM
```

The macro can be called using the macro quote characters:

```
         macld  <R6, 1>
         END
```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 16.

Predefined macro symbols

The symbol `_args` is set to the number of arguments passed to the macro. The following example shows how `_args` can be used:

```
        MODULE  ASAM8_MAN

        EXTERN  sub1

        MACRO DO_SUB1
          IF _args == 2
            CP    \0, \1
            JP    Z,  nocall
            CALL  sub1
nocall:
          ELSE
            CALL  sub1
          ENDIF
        ENDM

        RSEG  CODE

        DO_SUB1
        DO_SUB1  R6, #2

        END
```

Import the target-specific file L__args.fm here:

The following listing is generated:

```
     1    0000                      MODULE  ASAM8_MAN
     2    0000
     3    0000                      EXTERN  sub1
     4    0000
    15    0000
    16    0000                      RSEG  CODE
    17    0000
    18    0000                      DO_SUB1
    18.1  0000                        IF _args == 2
    18.2  0000                          CP    ,
    18.3  0000                          JP    Z,  nocall
    18.4  0000                          CALL  sub1
    18.5  0000          nocall:
    18.6  0000                        ELSE
    18.7  0000 F6....                   CALL  sub1
    18.8  0003                        ENDIF
    18.9  0003                      ENDM
    19    0003                      DO_SUB1  R6, #2
```

```
19.1  0003                          IF _args == 2
19.2  0003 A6C602                     CP    R6, #2
19.3  0006 6D....                     JP    Z,  nocall
19.4  0009 F6....                     CALL  sub1
19.5  000C          nocall:
19.6  000C                          ELSE
19.7  000C                            CALL  sub1
19.8  000C                          ENDIF
19.9  000C                        ENDM
20    000C
21    000C                        END
```

Use the `EXITM` directive to generate a premature exit from a macro.

`EXITM` is not allowed inside `REPT` ... `ENDR`, `REPTC` ... `ENDR`, or `REPTI` ... `ENDR`.

Use `LOCAL` to create symbols local to a macro. The `LOCAL` directive must be used before the symbol is used.

Each time a macro is expanded, new instances of local symbols are created by the `LOCAL` directive, so it is legal to use local symbols in recursive macros.

It is illegal to *redefine* a macro.

Passing special characters

Macro arguments that include commas or white space can be forced to be interpreted as one argument by using the matching quote characters `<` and `>` in the macro call.

For example:

```
macld   MACRO   op
        LD      op
        ENDM
```

It could be called using:

```
        macld   <R6, 1>
        END
```

You can redefine the macro quote characters with the `-M` command line option.

How macros are processed

There are three distinct phases in the macro process:

- The assembler performs scanning and saving of macro definitions. The text between `MACRO` and `ENDM` is saved but not syntax checked. Include-file references `$file` are recorded and will be included during macro *expansion*.

- A macro call forces the assembler to invoke the macro processor (expander). The macro expander switches (if not already in a macro) the assembler input stream from a source file to the output from the macro expander. The macro expander takes its input from the requested macro definition.

  The macro expander has no knowledge of assembler symbols since it only deals with text substitutions at source level. Before a line from the called macro definition is handed over to the assembler, the expander scans the line for all occurrences of symbolic macro arguments, and replaces them with their expansion arguments.

- The expanded line is then processed as any other assembler source line. The input stream to the assembler will continue to be the output from the macro processor, until all lines of the current macro definition have been read.

Repeating statements

Use the `REPT...ENDR` structure to assemble the same block of instructions a number of times. If *`expr`* evaluates to 0 nothing will be generated.

Use `REPTC` to assemble a block of instructions once for each character in a string. If the string contains a comma it should be enclosed in quotation marks.

Only double quotes have a special meaning and their only use is to enclose the characters to iterate over. Single quotes have no special meaning and are treated as any ordinary character.

Use `REPTI` to assemble a block of instructions once for each string in a series of strings. Strings containing commas should be enclosed in quotation marks.

EXAMPLES

This section gives examples of the different ways in which macros can make assembler programming easier.

Coding in-line for efficiency

In time-critical code it is often desirable to code routines in-line to avoid the overhead of a subroutine call and return. Macros provide a convenient way of doing this.

The following example outputs bytes from a buffer to a port:

```
        NAME    play

portb   SET     0x18
        RSEG    DATA
buffer  DS8     256

        RSEG    CODE
```

```
play    LD      R6, #HIGH(buffer)
        LD      R7, #LOW(buffer)
        LD      R5, 255
loop    LDEI    R0, @RR6
        LD      portb, R0
        DEC     R5
        JR      NE, loop
        RET

        END
```

The main program calls this routine as follows:

```
doplay  CALL    play
```

For efficiency we can recode this as the following macro:

```
        NAME    play

portb   SET     0x18
        RSEG    DATA
buffer  DS8     256

play    MACRO
        LOCAL   loop
        LD      R6, #HIGH(buffer)
        LD      R7, #LOW(buffer)
        LD      R5, 255
loop    LDEI    R0, @RR6
        LD      portb, R0
        DEC     R5
        JR      NE, loop
        ENDM

        RSEG    CODE
        play
        END
```

Notice the use of the `LOCAL` directive to make the label `loop` local to the macro; otherwise an error will be generated if the macro is used twice, as the `loop` label will already exist.

Using REPTC and REPTI

The following example assembles a series of calls to a subroutine `plot` to plot each character in a string:

```
         NAME    reptc

         EXTERN  plotc
banner   REPTC   chr, "Welcome"
         LD      R6, #'chr'
         CALL    plotc
         ENDR

         END
```

This produces the following code:

```
1    0000                       NAME    reptc
2    0000
3    0000                       EXTERN  plotc
4    0000               banner  REPTC   chr, "Welcome"
5    0000                       LD      R6, #'chr'
6    0000                       CALL    plotc
7    0000                       ENDR
7.1  0000 6C57                  LD      R6, #'W'
7.2  0002 F6..                  CALL    plotc
7.3  0005 6C65                  LD      R6, #'e'
7.4  0007 F6....                CALL    plotc
7.5  000A 6C6C                  LD      R6, #'l'
7.6  000C F6....                CALL    plotc
7.7  000F 6C63                  LD      R6, #'c'
7.8  0011 F6....                CALL    plotc
7.9  0014 6C6F                  LD      R6, #'o'
7.10 0016 F6....                CALL    plotc
7.11 0019 6C6D                  LD      R6, #'m'
7.12 001B F6....                CALL    plotc
7.13 001E 6C65                  LD      R6, #'e'
7.14 0020 F6....                CALL    plotc
8    0023
9    0023                       END
```

The following example uses `REPTI` to clear a number of memory locations:

```
         NAME    repti

         EXTERN  base, count, init

banner   REPTI   adds, base, count, init
         LD      R11, #LOW(adds)
```

```
LD      R10, #HIGH(adds)
LD      R6, #0
LDE     @RR8, R6
ENDR

END
```

This produces the following code:

```
1     0000                        NAME    repti
2     0000
3     0000                        EXTERN  base, count, init
4     0000
5     0000                banner  REPTI   adds, base, count, init
6     0000                        LD      R11, #LOW(adds)
7     0000                        LD      R10, #HIGH(adds)
8     0000                        LD      R6, #0
9     0000                        LDE     @RR8, R6
10    0000                        ENDR
10.1  0000 BC..                   LD      R11, #LOW( base)
10.2  0002 AC..                   LD      R10, #HIGH( base)
10.3  0004 6C00                   LD      R6, #0
10.4  0006 D369                   LDE     @RR8, R6
10.5  0008 BC..                   LD      R11, #LOW( count)
10.6  000A AC..                   LD      R10, #HIGH( count)
10.7  000C 6C00                   LD      R6, #0
10.8  000E D3969                  LDE     @RR8, R6
10.9  0010 BC..                   LD      R11, #LOW( init)
10.10 0012 AC..                   LD      R10, #HIGH( init)
10.11 0014 6C00                   LD      R6, #0
10.12 0016 D369                   LDE     @RR8, R6
11    0018
12    0018                        END
```

Listing control directives

These directives provide control over the assembler list file.

Directive	Description
COL	Sets the number of columns per page.
LSTCND	Controls conditional assembly listing.
LSTCOD	Controls multi-line code listing.
LSTEXP	Controls the listing of macro-generated lines.
LSTMAC	Controls the listing of macro definitions.

Table 23: Listing control directives

Directive	Description
LSTOUT	Controls assembler-listing output.
LSTPAG	Controls the formatting of output into pages.
LSTREP	Controls the listing of lines generated by repeat directives.
LSTXRF	Generates a cross-reference table.
PAGE	Generates a new page.
PAGSIZ	Sets the number of lines per page.

Table 23: Listing control directives (Continued)

SYNTAX

```
COL columns
LSTCND{+ | -}
LSTCOD{+ | -}
LSTEXP{+ | -}
LSTMAC{+ | -}
LSTOUT{+ | -}
LSTPAG{+ | -}
LSTREP{+ | -}
LSTXRF{+ | -}
PAGE
PAGSIZ lines
```

PARAMETERS

columns	An absolute expression in the range 80 to 132, default is 80
lines	An absolute expression in the range 10 to 150, default is 44

DESCRIPTION

Turning the listing on or off

Use `LSTOUT-` to disable all list output except error messages. This directive overrides all other listing control directives.

The default is `LSTOUT+`, which lists the output (if a list file was specified).

Listing conditional code and strings

Use `LSTCND+` to force the assembler to list source code only for the parts of the assembly that are not disabled by previous conditional `IF` statements, `ELSE`, or `END`.

The default setting is `LSTCND-`, which lists all source lines.

Use LSTCOD- to restrict the listing of output code to just the first line of code for a source line.

The default setting is LSTCOD+, which lists more than one line of code for a source line, if needed; i.e. long ASCII strings will produce several lines of output. Code generation is *not* affected.

Controlling the listing of macros

Use LSTEXP- to disable the listing of macro-generated lines. The default is LSTEXP+, which lists all macro-generated lines.

Use LSTMAC+ to list macro definitions. The default is LSTMAC-, which disables the listing of macro definitions.

Controlling the listing of generated lines

Use LSTREP- to turn off the listing of lines generated by the directives REPT, REPTC, and REPTI.

The default is LSTREP+, which lists the generated lines.

Generating a cross-reference table

Use LSTXRF+ to generate a cross-reference table at the end of the assembler list for the current module. The table shows values and line numbers, and the type of the symbol.

The default is LSTXRF-, which does not give a cross-reference table.

Specifying the list file format

Use COL to set the number of columns per page of the assembler list. The default number of columns is 80.

Use PAGSIZ to set the number of printed lines per page of the assembler list. The default number of lines per page is 44.

Use LSTPAG+ to format the assembler output list into pages.

The default is LSTPAG-, which gives a continuous listing.

Use PAGE to generate a new page in the assembler list file if paging is active.

EXAMPLES

Turning the listing on or off

To disable the listing of a debugged section of program:

```
LSTOUT-
; Debugged section
LSTOUT+
 ; Not yet debugged
```

Listing conditional code and strings

The following example shows how `LSTCND+` hides a call to a subroutine that is disabled by an `IF` directive:

```
        NAME    lstcndtst
        EXTERN  print

        RSEG    prom

debug   VAR     0
begin   IF      debug
        CALL    print
        ENDIF

        LSTCND+
begin2  IF      debug
        CALL    print
        ENDIF

        END
```

This will generate the following listing:

```
     1    0000                          NAME    lstcndtst
     2    0000                          EXTERN  print
     3    0000
     4    0000                          RSEG    prom
     5    0000                  debug   VAR     0
     6    0000                  begin   IF      debug
     7    0000                          CALL    print
     8    0000                          ENDIF
     9    0000
    10    0000                          LSTCND+
    11    0000                  begin2  IF      debug
    12    0000                          ENDIF
    13    0000                          END
```

The following example shows the effect of `LSTCOD+` on the code generated by a `DC16` directive:

```
 1    0000                    NAME    lstcodtst
 2    0000 0001000A           DC16    1,10,100,100,1000
 3    000A
 4    000A                    LSTCOD+
 5    000A 0001000A           DC16    1,10,100,100,1000
           00640064
           03E8
 6    0014
 7    0014                    END
```

Controlling the listing of macros

The following example shows the effect of `LSTMAC` and `LSTEXP`:

```
dec2    MACRO  arg
        DEC    arg
        DEC    arg
        ENDM

        LSTMAC-

inc2    MACRO  arg
        INC    arg
        INC    arg
        ENDM

begin   dec2   R6

        LSTEXP-
        inc2   R7
        RET

        END    begin
```

This will produce the following output:

```
  5    0000
  6    0000                         LSTMAC-
  7    0000
 12    0000                 begin   dec2    R6
 13    0000                 begin   dec2    R6
 13.1  0000 00C6                    DEC     R6
 13.2  0002 00C6                    DEC     R6
 13.3  0004                         ENDM
 14    0004
 15    0004                         LSTEXP-
```

```
16    0004                    inc2    R7
17    0006 AF                 RET
18    0007
19    0007                    END     begin
```

Formatting listed output

The following example formats the output into pages of 66 lines each with 132 columns. The `LSTPAG` directive organizes the listing into pages, starting each module on a new page. The `PAGE` directive inserts additional page breaks.

```
PAGSIZ 66  ; Page size
COL 80
LSTPAG+
...
ENDMOD
MODULE
...
PAGE
...
```

C-style preprocessor directives

The following C-language preprocessor directives are available:

Directive	Description
`#define`	Assigns a value to a label.
`#elif`	Introduces a new condition in a `#if...#endif` block.
`#else`	Assembles instructions if a condition is false.
`#endif`	Ends a `#if`, `#ifdef`, or `#ifndef` block.
`#error`	Generates an error.
`#if`	Assembles instructions if a condition is true.
`#ifdef`	Assembles instructions if a symbol is defined.
`#ifndef`	Assembles instructions if a symbol is undefined.
`#include`	Includes a file.
`#pragma`	Recognized and ignored.
`#undef`	Undefines a label.
`/*comment*/`	C-style comment delimiter.
`//`	C++style comment delimiter.

Table 24: C-style preprocessor directives

SYNTAX

```
#define label text
#elif condition
#else
#endif
#error "message"
#if condition
#ifdef label
#ifndef label
#include {"filename" | <filename>}
#message "message"
#undef label
/*comment*/
//comment
```

PARAMETERS

condition	One of the following:	
	An absolute expression	The expression must not contain forward or external references, and any non-zero value is considered as true.
	string1=string	The condition is true if *string1* and *string2* have the same length and contents.
	string1<>string2	The condition is true if *string1* and *string2* have different length or contents.
filename	Name of file to be included.	
label	Symbol to be defined, undefined, or tested.	
message	Text to be displayed.	
text	Value to be assigned.	

DESCRIPTION

Defining and undefining labels

Use `#define` to define a temporary label.

`#define` *`label value`*

is similar to:

`label` `VAR` *`value`*

Use `#undef` to undefine a label; the effect is as if it had not been defined.

Use `/*...*/` to comment sections of the assembler listing.

Use `//` to mark the rest of the line as comment.

Conditional directives

Use the `#if`...`#else`...`#endif` directives to control the assembly process at assembly time. If the condition following the `#if` directive is not true, the subsequent instructions will not generate any code (i.e. it will not be assembled or syntax checked) until a `#endif` or `#else` directive is found.

All assembler directives (except for `END`) and file inclusion may be disabled by the conditional directives. Each `#if` directive must be terminated by a `#endif` directive. The `#else` directive is optional and, if used, it must be inside a `#if`...`#endif` block.

`#if`...`#endif` and `#if`...`#else`...`#endif` blocks may be nested to any level.

Use `#ifdef` to assemble instructions up to the next `#else` or `#endif` directive only if a symbol is defined.

Use `#ifndef` to assemble instructions up to the next `#else` or `#endif` directive only if a symbol is undefined.

Including source files

Use `#include` to insert the contents of a file into the source file at a specified point.

`#include "`*`filename`*`"` searches the following directories in the specified order:

1 The source file directory.

2 The directories specified by the `-I` option, or options.

3 The current directory.

`#include <filename>` searches the following directories in the specified order:

1 The directories specified by the `-I` option, or options.

2 The current directory.

Displaying errors

Use `#error` to force the assembler to generate an error, such as in a user-defined test.

Defining comments

Use `/* ... */` to comment sections of the assembler listing.

Use `//` to mark the rest of the line as comment.

Note: It is important to avoid mixing the assembler language with the C-style preprocessor directives. Conceptually, they are different languages and mixing them may lead to unexpected behavior, since an assembler directive is not necessarily accepted as a part of the C language.

The following example illustrates some problems that may occur when assembler comments are used in the C-style preprocessor:

```
#define   five 5 ; comment

     LD    five,  #3       ; syntax error
     ; Expands to "LD 0x05 ; comment, #3"

     LD    R3,  #five + adde ; incorrect code
     ; Expands to "LD R3, 0x05 ; comment + addr"
```

EXAMPLES

Using conditional directives

The following example defines the labels `tweak` and `adjust`. If `adjust` is defined, then register 16 is decremented by an amount that depends on `adjust`, in this case 30.

```
#define tweak 1
#define adjust 3

#ifdef  tweak
#if     adjust=1
        SUB     R6,#4
#elif   adjust=2
        SUB     R6,#20
#elif   adjust=3
        SUB     R6,#30
```

```
#endif
#endif                              /* ifdef tweak */
```

Including a source file

The following example uses `#include` to include a file defining macros into the source file. For example, the following macros could be defined in `Macros.s18`:

```
xch     MACRO   a,b
        PUSH    a
        MOV     a,b
        POP     b
        ENDM
```

The macro definitions can then be included, using `#include`, as in the following example:

```
        NAME   include

; Standard macro definitions
#include c:\iar\asm\inc\macros.s18"

; Program
main:   xch     R6,R7
        RET
        END     main
```

Space allocation directives

These directives define temporary values or reserve memory:

Directive	Description	Expression restrictions
`DC8, DB`	Generates 8-bit constants, including strings.	
`DC16, DW`	Generates 16-bit constants.	
`DC24, DP`	Generates 24-bit constants.	
`DC32, DD`	Generates 32-bit constants.	
`DS8, DS`	Reserves memory space without initializing (8-bit).	No external references Absolute

Table 25: Space allocation directives

SYNTAX

```
DC8 expr [,expr] ...
DC16 expr [,expr] ...
DC24 expr [,expr] ...
DC32 expr [,expr] ...
DB expr [,expr] ...
DW expr [,expr] ...
DP expr [,expr] ...
DD expr [,expr] ...
DS8 expr
DS expr
```

PARAMETERS

expr	A valid absolute, relocatable, or external expression, or an ASCII string. ASCII strings will be zero filled to a multiple of the data size implied by the directive. Double-quoted strings will be zero-terminated.
value	A valid absolute expression or a floating-point constant.

DESCRIPTIONS

Use the data definition and allocation directives according to the following table; it shows which directives reserve and initialize memory space or reserve uninitialized memory space, and their size.

Size	Reserve and initialize memory	Reserve unitialized memory
8-bit integers	DC8, DB	DS8, DS
16-bit integers	DC16, DW	
24-bit integers	DC24, DP	
32-bit integers	DC32, DD	

Table 26: Using data definition or allocation directives

EXAMPLES

Generating lookup table

The following example generates a lookup table of addresses to routines:

```
        NAME    table
        RSEG    CONST
table   DC16    addsubr/2, subsubr/2, clrsubr/2
        RSEG    CODE
addsubr ADD     R6,R7
        RET
```

```
subsubr SUB     R6,R7
        RET
clrsubr CLR     R6
        RET

        END
```

Defining strings

To define a string:

```
mymsg   DC8 'Please enter your name'
```

To define a string which includes a trailing zero:

```
myCstr  DC8 "This is a string."
```

To include a single quote in a string, enter it twice; for example:

```
errmsg  DC8 'Don''t understand!'
```

Reserving space

To reserve space for `0xA` bytes:

```
table   DS8   0xA
```

Assembler control directives

These directives provide control over the operation of the assembler.

Directive	Description
`$`	Includes a file.
`CASEOFF`	Disables case sensitivity.
`CASEON`	Enables case sensitivity.
`RADIX`	Sets the default base.

Table 27: Assembler control directives

SYNTAX

```
$filename
CASEOFF
CASEON
RADIX expr
```

PARAMETERS

comment Comment ignored by the assembler.

expr Default base; default 10 (decimal).

filename Name of file to be included. The `$` character must be the first character on the line.

DESCRIPTION

Use `$` to insert the contents of a file into the source file at a specified point.

Use `RADIX` to set the default base for use in conversion of constants from ASCII source to the internal binary format.

Controlling case sensitivity

Use `CASEON` or `CASEOFF` to turn on or off case sensitivity for user-defined symbols. By default case sensitivity is off.

When `CASEOFF` is active all symbols are stored in upper case, and all symbols used by XLINK should be written in upper case in the XLINK definition file.

EXAMPLES

Including a source file

The following example uses `$` to include a file defining macros into the source file. For example, the following macros could be defined in `Mymacros.s18`:

```
xch     MACRO   a,b
        PUSH    a
        LD      a,b
        POP     b
        ENDM
```

The macro definitions can be included with a `$` directive, as in:

```
        NAME    include

; standard macro definitions

$mymacros.s18

; program
main
        xch     R6,R7
        RET
        END   main
```

Defining comments

The following example shows how `/*...*/` can be used for a multi-line comment:

```
/*
Program to read serial input.
Version 3: 19.12.01
Author: mjp
*/
```

Changing the base

To set the default base to 16:

```
        RADIX  D'16
        LD     R6,#12
```

The immediate argument will then be interpreted as `H'12`.

To change the base from 16 to 10, *expr* must be written in hexadecimal format, for example:

```
RADIX  0x0A
```

Controlling case sensitivity

When `CASEOFF` is set, `label` and `LABEL` are identical in the following example:

```
label   NOP       ; Stored as "LABEL"
        JP        LABEL
```

The following will generate a duplicate label error:

```
        CASEOFF

label   NOP
LABEL   NOP       ; Error, "LABEL" already defined

        END
```

Compiler function directives

The following directives are used by the C compiler:

Directive	Description
ARGFRAME	Defines a function's arguments.
FUNCALL	Defines function call information.
FUNCTION	Defines a function.
LOCFRAME	Defines a function's local variables.

Table 28: Compiler function directives

DESCRIPTION

The compiler function directives can be used by the compiler to pass information about functions to the linker. These directives are normally not used in assembler programming. For information on how to use these directives, see the chapter *Assembler language interface* in the *SAM8 IAR C Compiler Reference Guide*.

Call frame information directives

These directives allow backtrace information to be defined.

Directive	Description
CFI BASEADDRESS	Declares a base address CFA (Canonical Frame Address).
CFI BLOCK	Starts a data block.
CFI CODEALIGN	Declares code alignment.
CFI COMMON	Starts or extends a common block.
CFI CONDITIONAL	Declares data block to be a conditional thread.
CFI DATAALIGN	Declares data alignment.
CFI ENDBLOCK	Ends a data block.
CFI ENDCOMMON	Ends a common block.
CFI ENDNAMES	Ends a names block.
CFI FRAMECELL	Creates a reference into the caller's frame.
CFI FUNCTION	Declares a function associated with data block.
CFI INVALID	Starts range of invalid backtrace information.
CFI NAMES	Starts a names block.
CFI NOFUNCTION	Declares data block to not be associated with a function.

Table 29: Call frame information directives

Directive	Description
CFI PICKER	Declares data block to be a picker thread.
CFI REMEMBERSTATE	Remembers the backtrace information state.
CFI RESOURCE	Declares a resource.
CFI RESOURCEPARTS	Declares a composite resource.
CFI RESTORESTATE	Restores the saved backtrace information state.
CFI RETURNADDRESS	Declares a return address column.
CFI STACKFRAME	Declares a stack frame CFA.
CFI STATICOVERLAYFRAME	Declares a static overlay frame CFA.
CFI VALID	Ends range of invalid backtrace information.
CFI VIRTUALRESOURCE	Declares a virtual resource.
CFI *cfa*	Declares the value of a CFA.
CFI *resource*	Declares the value of a resource.

Table 29: Call frame information directives (Continued)

SYNTAX

The syntax definitions below show the syntax of each directive. The directives are grouped according to usage.

Names block directives

```
CFI NAMES name
CFI ENDNAMES name
CFI RESOURCE resource : bits [, resource : bits] …
CFI VIRTUALRESOURCE resource : bits [, resource : bits] …
CFI RESOURCEPARTS resource part, part [, part] …
CFI STACKFRAME cfa resource type [, cfa resource type] …
CFI STATICOVERLAYFRAME cfa segment [, cfa segment] …
CFI BASEADDRESS cfa type [, cfa type] …
```

Extended names block directives

```
CFI NAMES name EXTENDS namesblock
CFI ENDNAMES name
CFI FRAMECELL cell cfa (offset): size [, cell cfa (offset): size] …
```

Common block directives

```
CFI COMMON name USING namesblock
CFI ENDCOMMON name
CFI CODEALIGN codealignfactor
```

```
CFI DATAALIGN dataalignfactor
CFI RETURNADDRESS resource type
CFI cfa { NOTUSED | USED }
CFI cfa { resource | resource + constant | resource - constant }
CFI cfa cfiexpr
CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
CFI resource { resource | FRAME(cfa, offset) }
CFI resource cfiexpr
```

Extended common block directives

```
CFI COMMON name EXTENDS commonblock USING namesblock
CFI ENDCOMMON name
```

Data block directives

```
CFI BLOCK name USING commonblock
CFI ENDBLOCK name
CFI { NOFUNCTION | FUNCTION label }
CFI { INVALID | VALID }
CFI { REMEMBERSTATE | RESTORESTATE }
CFI PICKER
CFI CONDITIONAL label [, label] …
CFI cfa { resource | resource + constant | resource - constant }
CFI cfa cfiexpr
CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
CFI resource { resource | FRAME(cfa, offset) }
CFI resource cfiexpr
```

PARAMETERS

bits	The size of the resource in bits.
cell	The name of a frame cell.
cfa	The name of a CFA (canonical frame address).
cfiexpr	A CFI expression (see *CFI expressions*, page 88).
codealignfactor	The smallest factor of all instruction sizes. Each CFI directive for a data block must be placed according to this alignment. 1 is the default and can always be used, but a larger value will shrink the produced backtrace information in size. The possible range is 1–256.
commonblock	The name of a previously defined common block.

constant	A constant value or an assembler expression that can be evaluated to a constant value.
dataalignfactor	The smallest factor of all frame sizes. If the stack grows towards higher addresses, the factor is negative; if it grows towards lower addresses, the factor is positive. 1 is the default, but a larger value will shrink the produced backtrace information in size. The possible ranges are -256 – -1 and 1 – 256.
label	A function label.
name	The name of the block.
namesblock	The name of a previously defined names block.
offset	The offset relative the CFA. An integer with an optional sign.
part	A part of a composite resource. The name of a previously declared resource.
resource	The name of a resource.
segment	The name of a segment.
size	The size of the frame cell in bytes.
type	The memory type, such as `CODE`, `CONST` or `DATA`. In addition, any of the memory types supported by the IAR XLINK Linker. It is used solely for the purpose of denoting an address space.

DESCRIPTIONS

The Call Frame Information directives (CFI directives) are an extension to the debugging format of the IAR C-SPY Debugger. The CFI directives are used for defining the *backtrace information* for the instructions in a program. The compiler normally generates this information, but for library functions and other code written purely in assembler language, backtrace information has to be added if you want to use the call frame stack in the debugger.

The backtrace information is used to keep track of the contents of *resources*, such as registers or memory cells, in the assembler code. This information is used by the IAR C-SPY Debugger to go “back” in the call stack and show the correct values of registers or other resources before entering the function. In contrast with traditional approaches, this permits the debugger to run at full speed until it reaches a breakpoint, stop at the breakpoint, and retrieve backtrace information at that point in the program. The information can then be used to compute the contents of the resources in any of the calling functions—assuming they have call frame information as well.

Backtrace rows and columns

At each location in the program where it is possible for the debugger to break execution, there is a *backtrace row*. Each backtrace row consists of a set of *columns*, where each column represents an item that should be tracked. There are three kinds of columns:

- The *resource columns* keep track of where the original value of a resource can be found.
- The canonical frame address columns (*CFA columns*) keep track of the top of the function frames.
- The *return address column* keeps track of the location of the return address.

There is always exactly one return address column and usually only one CFA column, although there may be more than one.

Defining a names block

A *names block* is used to declare the resources available for a processor. Inside the names block, all resources that can be tracked are defined.

Start and end a names block with the directives:

```
CFI NAMES name
CFI ENDNAMES name
```

where `name` is the name of the block.

Only one names block can be open at a time.

Inside a names block, four different kinds of declarations may appear: a resource declaration, a stack frame declaration, a static overlay frame declaration, or a base address declaration:

- To declare a resource, use one of the directives:

  ```
  CFI RESOURCE resource : bits
  CFI VIRTUALRESOURCE resource : bits
  ```

  The parameters are the name of the resource and the size of the resource in bits. A virtual resource is a logical concept, in contrast to a “physical” resource such as a processor register. Virtual resources are usually used for the return address.

  More than one resource can be declared by separating them with commas.

  A resource may also be a composite resource, made up of at least two parts. To declare the composition of a composite resource, use the directive:

  ```
  CFI RESOURCEPARTS resource part, part, …
  ```

  The parts are separated with commas. The resource and its parts must have been previously declared as resources, as described above.

- To declare a stack frame CFA, use the directive:

  `CFI STACKFRAME` *`cfa resource type`*

  The parameters are the name of the stack frame CFA, the name of the associated resource (the stack pointer), and the segment type (to get the address space). More than one stack frame CFA can be declared by separating them with commas.

  When going "back" in the call stack, the value of the stack frame CFA is copied into the associated stack pointer resource to get a correct value for the previous function frame.

- To declare a static overlay frame CFA, use the directive:

  `CFI STATICOVERLAYFRAME` *`cfa segment`*

  The parameters are the name of the CFA and the name of the segment where the static overlay for the function is located. More than one static overlay frame CFA can be declared by separating them with commas.

- To declare a base address CFA, use the directive:

  `CFI BASEADDRESS` *`cfa type`*

  The parameters are the name of the CFA and the segment type. More than one base address CFA can be declared by separating them with commas.

  A base address CFA is used to conveniently handle a CFA. In contrast to the stack frame CFA, there is no associated stack pointer resource to restore.

Extending a names block

In some special cases you have to extend an existing names block with new resources. This occurs whenever there are routines that manipulate call frames other than their own, such as routines for handling, entering, and leaving C functions; these routines manipulate the caller's frame. Extended names blocks are normally used only by compiler developers.

Extend an existing names block with the directive:

`CFI NAMES` *`name`* `EXTENDS` *`namesblock`*

where *`namesblock`* is the name of the existing names block and *`name`* is the name of the new extended block. The extended block must end with the directive:

`CFI ENDNAMES` *`name`*

Defining a common block

The *common block* is used for declaring the initial contents of all tracked resources. Normally, there is one common block for each calling convention used.

Start a common block with the directive:

```
CFI COMMON name USING namesblock
```

where *name* is the name of the new block and *namesblock* is the name of a previously defined names block.

Declare the return address column with the directive:

```
CFI RETURNADDRESS resource type
```

where *resource* is a resource defined in *namesblock* and *type* is the segment type. You have to declare the return address column for the common block.

End a common block with the directive:

```
CFI ENDCOMMON name
```

where *name* is the name used to start the common block.

Inside a common block you can declare the initial value of a CFA or a resource by using the directives listed last in *Common block directives*, page 80. For more information on these directives, see *Simple rules*, page 86, and *CFI expressions*, page 88.

Extending a common block

Since you can extend a names block with new resources, it is necessary to have a mechanism for describing the initial values of these new resources. For this reason, it is also possible to extend common blocks, effectively declaring the initial values of the extra resources while including the declarations of another common block. Just as in the case of extended names blocks, extended common blocks are normally only used by compiler developers.

Extend an existing common block with the directive:

```
CFI COMMON name EXTENDS commonblock USING namesblock
```

where *name* is the name of the new extended block, *commonblock* is the name of the existing common block, and *namesblock* is the name of a previously defined names block. The extended block must end with the directive:

```
CFI ENDCOMMON name
```

Defining a data block

The *data block* contains the actual tracking information for one continuous piece of code. No segment control directive may appear inside a data block.

Start a data block with the directive:

```
CFI BLOCK name USING commonblock
```

where *name* is the name of the new block and *commonblock* is the name of a previously defined common block.

If the piece of code is part of a defined function, specify the name of the function with the directive:

```
CFI FUNCTION label
```

where *label* is the code label starting the function.

If the piece of code is not part of a function, specify this with the directive:

```
CFI NOFUNCTION
```

End a data block with the directive:

```
CFI ENDBLOCK name
```

where *name* is the name used to start the data block.

Inside a data block you may manipulate the values of the columns by using the directives listed last in *Data block directives*, page 81. For more information on these directives, see *Simple rules*, page 86, and *CFI expressions*, page 88.

SIMPLE RULES

To describe the tracking information for individual columns, there is a set of simple rules with specialized syntax:

```
CFI cfa { NOTUSED | USED }
CFI cfa { resource | resource + constant | resource - constant }
CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
CFI resource { resource | FRAME(cfa, offset) }
```

These simple rules can be used both in common blocks to describe the initial information for resources and CFAs, and inside data blocks to describe changes to the information for resources or CFAs.

In those rare cases where the descriptive power of the simple rules are not enough, a full CFI expression can be used to describe the information (see *CFI expressions*, page 88). However, whenever possible, you should always use a simple rule instead of a CFI expression.

There are two different sets of simple rules: one for resources and one for CFAs.

Simple rules for resources

The rules for resources conceptually describe where to find a resource when going back one call frame. For this reason, the item following the resource name in a CFI directive is referred to as the *location* of the resource.

To declare that a tracked resource is restored, that is, already correctly located, use SAMEVALUE as the location. Conceptually, this declares that the resource does not have to be restored since it already contains the correct value. For example, to declare that a register REG is restored to the same value, use the directive:

```
CFI REG SAMEVALUE
```

To declare that a resource is not tracked, use UNDEFINED as location. Conceptually, this declares that the resource does not have to be restored (when going back one call frame) since it is not tracked. Usually it is only meaningful to use it to declare the initial location of a resource. For example, to declare that REG is a scratch register and does not have to be restored, use the directive:

```
CFI REG UNDEFINED
```

To declare that a resource is temporarily stored in another resource, use the resource name as its location. For example, to declare that a register REG1 is temporarily located in a register REG2 (and should be restored from that register), use the directive:

```
CFI REG1 REG2
```

To declare that a resource is currently located somewhere on the stack, use FRAME(*cfa*, *offset*) as location for the resource, where *cfa* is the CFA identifier to use as "frame pointer" and *offset* is an offset relative the CFA. For example, to declare that a register REG is located at offset -4 counting from the frame pointer CFA_SP, use the directive:

```
CFI REG FRAME(CFA_SP,-4)
```

For a composite resource there is one additional location, CONCAT, which declares that the location of the resource can be found by concatenating the resource parts for the composite resource. For example, consider a composite resource RET with resource parts RETLO and RETHI. To declare that the value of RET can be found by investigating and concatenating the resource parts, use the directive:

```
CFI RET CONCAT
```

This requires that at least one of the resource parts has a definition, using the rules described above.

Simple rules for CFAs

In contrast with the rules for resources, the rules for CFAs describe the address of the beginning of the call frame. The call frame often includes the return address pushed by the subroutine calling instruction. The CFA rules describe how to compute the address to the beginning of the current call frame. There are two different forms of CFAs, stack frames and static overlay frames, each declared in the associated names block. See *Names block directives*, page 80.

Each stack frame CFA is associated with a resource, such as the stack pointer. When going back one call frame the associated resource is restored to the current CFA. For stack frame CFAs there are two possible simple rules: an offset from a resource (not necessarily the resource associated with the stack frame CFA) or `NOTUSED`.

To declare that a CFA is not used, and that the associated resource should be tracked as a normal resource, use `NOTUSED` as the address of the CFA. For example, to declare that the CFA with the name `CFA_SP` is not used in this code block, use the directive:

```
CFI CFA_SP NOTUSED
```

To declare that a CFA has an address that is offset relative the value of a resource, specify the resource and the offset. For example, to declare that the CFA with the name `CFA_SP` can be obtained by adding 4 to the value of the `SP` resource, use the directive:

```
CFI CFA_SP SP + 4
```

For static overlay frame CFAs, there are only two possible declarations inside common and data blocks: `USED` and `NOTUSED`.

CFI EXPRESSIONS

Call Frame Information expressions (CFI expressions) can be used when the descriptive power of the simple rules for resources and CFAs is not enough. However, you should always use a simple rule when one is available.

CFI expressions consist of operands and operators. Only the operators described below are allowed in a CFI expression. In most cases, they have an equivalent operator in the regular assembler expressions.

In the operand descriptions, *`cfiexpr`* denotes one of the following:

- A CFI operator with operands
- A numeric constant
- A CFA name
- A resource name.

Unary operators

Overall syntax: `OPERATOR(operand)`

Operator	Operand	Description
UMINUS	*cfiexpr*	Performs arithmetic negation on a CFI expression.
NOT	*cfiexpr*	Negates a logical CFI expression.
COMPLEMENT	*cfiexpr*	Performs a bitwise NOT on a CFI expression.
LITERAL	*expr*	Get the value of the assembler expression. This can insert the value of a regular assembler expression into a CFI expression.

Table 30: Unary operators in CFI expressions

Binary operators

Overall syntax: `OPERATOR(operand1,operand2)`

Operator	Operands	Description
ADD	*cfiexpr,cfiexpr*	Addition
SUB	*cfiexpr,cfiexpr*	Subtraction
MUL	*cfiexpr,cfiexpr*	Multiplication
DIV	*cfiexpr,cfiexpr*	Division
MOD	*cfiexpr,cfiexpr*	Modulo
AND	*cfiexpr,cfiexpr*	Bitwise AND
OR	*cfiexpr,cfiexpr*	Bitwise OR
XOR	*cfiexpr,cfiexpr*	Bitwise XOR
EQ	*cfiexpr,cfiexpr*	Equal
NE	*cfiexpr,cfiexpr*	Not equal
LT	*cfiexpr,cfiexpr*	Less than
LE	*cfiexpr,cfiexpr*	Less than or equal
GT	*cfiexpr,cfiexpr*	Greater than
GE	*cfiexpr,cfiexpr*	Greater than or equal
LSHIFT	*cfiexpr,cfiexpr*	Logical shift left of the left operand. The number of bits to shift is specified by the right operand. The sign bit will not be preserved when shifting.

Table 31: Binary operators in CFI expressions

Operator	Operands	Description
RSHIFTA	*cfiexpr,cfiexpr*	Arithmetic shift right of the left operand. The number of bits to shift is specified by the right operand. In contrast with RSHIFTL the sign bit will be preserved when shifting.
RSHIFTL	*cfiexpr,cfiexpr*	Logical shift right of the left operand. The number of bits to shift is specified by the right operand. The sign bit will not be preserved when shifting.

Table 31: Binary operators in CFI expressions (Continued)

Ternary operators

Overall syntax: `OPERATOR(operand1,operand2,operand3)`

Operator	Operands	Description
FRAME	*cfa,size,offset*	Get value from stack frame. The operands are: cfa An identifier denoting a previously declared CFA. sizeA constant expression denoting a size in bytes. offsetA constant expression denoting an offset in bytes. Gets the value at address *cfa+offset* of size *size*.
IF	*cond,true,false*	Conditional operator. The operands are: condA CFA expression denoting a condition. trueAny CFA expression. falseAny CFA expression. If the conditional expression is non-zero, the result is the value of the *true* expression; otherwise the result is the value of the *false* expression.
LOAD	*size,type,addr*	Get value from memory. The operands are: sizeA constant expression denoting a size in bytes. typeA memory type. addrA CFA expression denoting a memory address. Gets the value at address *addr* in segment type *type* of size *size*.

Table 32: Ternary operators in CFI expressions

EXAMPLE

The following is a generic example and not an example specific to the SAM8 microcontroller. This will simplify the example and clarify the usage of the CFI directives. A target-specific example can be obtained by generating assembler output when compiling a C source file.

Consider a generic processor with a stack pointer `SP`, and two registers `R0` and `R1`. Register `R0` will be used as a scratch register (the register is destroyed by the function call), whereas register `R1` has to be restored after the function call. For reasons of simplicity, all instructions, registers, and addresses will have a width of 16 bits.

Consider the following short code sample with the corresponding backtrace rows and columns. At entry, assume that the stack contains a 16-bit return address. The stack grows from high addresses towards zero. The CFA denotes the top of the call frame, that is, the value of the stack pointer after returning from the function.

Address	CFA	SP	R0	R1	RET	Assembler code
`0000`	SP + 2		—	SAME	CFA - 2	`func1: PUSH R1`
`0002`	SP + 4			CFA - 4		`MOV R1,#4`
`0004`						`CALL func2`
`0006`						`POP R0`
`0008`	SP + 2			R0		`MOV R1,R0`
`000A`				SAME		`RET`

Table 33: Code sample with backtrace rows and columns

Each backtrace row describes the state of the tracked resources *before* the execution of the instruction. As an example, for the `MOV R1,R0` instruction the original value of the `R1` register is located in the `R0` register and the top of the function frame (the CFA column) is `SP + 2`. The backtrace row at address `0000` is the initial row and the result of the calling convention used for the function.

The SP column is empty since the CFA is defined in terms of the stack pointer. The RET column is the return address column—that is, the location of the return address. The R0 column has a '—' in the first line to indicate that the value of `R0` is undefined and does not need to be restored on exit from the function. The R1 column has `SAME` in the initial row to indicate that the value of the `R1` register will be restored to the same value it already has.

Defining the names block

The names block for the small example above would be:

```
CFI NAMES trivialNames
CFI RESOURCE SP:16, R0:16, R1:8
CFI STACKFRAME CFA SP DATA

;; The virtual resource for the return address column
CFI VIRTUALRESOURCE RET:16
CFI ENDNAMES trivialNames
```

Defining the common block

The common block for the simple example above would be:

```
CFI COMMON trivialCommon USING trivialNames
CFI RETURNADDRESS RET DATA
CFI CFA SP + 2
CFI R0 UNDEFINED
CFI R1 SAMEVALUE
CFI RET FRAME(CFA,-2)  ; Offset -2 from top of frame
CFI ENDCOMMON trivialCommon
```

Note: `SP` may not be changed using a CFI directive since it is the resource associated with `CFA`.

Defining the data block

Continuing the simple example, the data block would be:

```
     RSEG   CODE:CODE
     CFI    BLOCK func1block USING trivialCommon
     CFI    FUNCTION func1
func1:
     PUSH   R1
     CFI    CFA SP + 4
     CFI    R1 FRAME(CFA,-4)
     LD     R1,#4
     CALL   func2
     POP    R0
     CFI    R1 R0
     CFI    CFA SP + 2
     LD     R1,R0
     CFI    R1 SAMEVALUE
     RET
     CFI ENDBLOCK func1block
```

Note that the CFI directives are placed *after* the instruction that affects the backtrace information.

Diagnostics

This chapter describes the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.

Message format

All diagnostic messages are issued as complete, self-explanatory messages. A typical diagnostic message from the assembler is produced in the form:

`filename,linenumber level[tag]: message`

where *filename* is the name of the source file in which the error was encountered; *linenumber* is the line number at which the assembler detected the error; *level* is the level of seriousness of the diagnostic; *tag* is a unique tag that identifies the diagnostic message; and *message* is a self-explanatory message, possibly several lines long.

Diagnostic messages are displayed on the screen, as well as printed in the optional list file.

Severity levels

The diagnostics are divided into different levels of severity:

Line error

A diagnostic message that is produced when the assembler finds an error in the parameters given on the command line. The assembler then issues a self-explanatory message.

Error

A diagnostic message that is produced when the assembler has found a construct which clearly violates the language rules, such that code cannot be produced.

Fatal error

A diagnostic message that is produced when the assembler has found a condition that not only prevents code generation, but which makes further processing of the source code pointless. After the diagnostic message has been issued, the assembly terminates.

Memory overflow

A diagnostic message that is produced when the assembler runs out of memory.

Internal error

A diagnostic message that is produced when a serious and unexpected failure occurs due to a fault in the assembler itself. After the diagnostic message has been issued, the assembly terminates.

Warning

A diagnostic message that is produced when the assembler finds a programming error or omission which likely to cause problems, but not so severe as to prevent the completion of the assembly. These warnings can be disabled by use of the command-line option `-w`.

A

B

C

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Symbols