H8 Assembler, Linker, and Librarian

Programming Guide

Command Line Version

COPYRIGHT NOTICE

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment on any part of IAR Systems. While the information contained herein is assumed to be accurate, IAR Systems assumes no responsibility for any errors or omissions.

In no event shall IAR Systems, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

TRADEMARKS

C-SPY is a trademark of IAR Systems. MS-DOS is a trademark of Microsoft Corp.

All other product names are trademarks or registered trademarks of their respective owners.

First edition: October 1996
Part no: AH8C–1

This documentation was produced by Human-Computer Interface.

WELCOME

Welcome to the H8 Assembler, Linker, and Librarian Programming Guide.

This guide provides reference information about the IAR Systems Assembler, XLINK Linker, and XLIB Librarian for the Hitachi H8 Series microprocessors.

Before reading this guide we recommend you refer to the *QuickStart Card*, or the chapter *Installation and documentation route map*, for information about installing the IAR Systems tools and an overview of the documentation.

Refer to the *H8 Command Line Interface Guide* for general information about running the IAR Systems tools from the command line, and a simple tutorial to illustrate how to use them.

For information about programming with the H8 C Compiler refer to the *H8 C Compiler Programming Guide*.

If your product includes the optional H8 C-SPY debugger refer to the *H8 C-SPY User Guide* for information about debugging with C-SPY.

ABOUT THIS GUIDE

This guide consists of the following parts and chapters:

Installation and documentation route map explains how to install and run the IAR Systems tools, and gives an overview of the documentation supplied with them.

H8 Assembler

The *Introduction* provides a brief overview of the H8 Assembler.

The *Tutorial* explains how to use the most important features of the assembler to develop machine-code programs. It also describes a typical development cycle using XLINK and XLIB.

Assembler options summary explains how to set the H8 Assembler options, and gives an alphabetical summary of the options.

Assembler options reference then gives reference information about each option.

Assembler file formats describes the source format for the H8 Assembler, and the format of assembler listings.

Assembler operator summary gives a summary of the assembler operators, arranged in order of precedence.

Assembler operator reference then gives a complete alphabetical list of the H8 Series Assembler operators, with a full description of each one.

Assembler directives reference gives complete reference information about the H8 Series directives, classified into groups according to their function.

Assembler instructions lists the H8 instruction mnemonics, with details of the addressing modes that can be used with each one.

XLINK Linker

XLINK Linker introduces the XLINK Linker, and describes the XLINK listing format.

XLINK options summary explains how to set the XLINK options, and gives an alphabetical summary of the options.

XLINK options reference then gives detailed information about each option.

XLINK output formats summarizes the output formats available from XLINK.

XLIB Librarian

XLIB Librarian introduces the XLIB Librarian, which is designed to allow you to create and maintain relocatable libraries of routines.

XLIB command summary gives a summary of the XLIB commands.

XLIB command reference then gives complete reference information about each XLIB command.

Diagnostics

Assembler diagnostics provides a list of error messages specific to the H8 Assembler.

XLINK diagnostics and *XLIB diagnostics* describe the error and warning messages produced by XLINK and XLIB, together with explanations and suggested courses of action in each case.

ASSUMPTIONS

This guide assumes that you already have a working knowledge of the following:

- The H8 Series processor you are using.
- The H8 Series assembler language.
- MS-DOS or UNIX, depending on your host system.

CONVENTIONS

This guide uses the following typographical conventions:

Style	*Used for*
`computer`	Text that you type in, or that appears on the screen.
`parameter`	A label representing the actual value you should type 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, dialog boxes, and windows that appear on the screen.
reference	A cross-reference to another part of this guide, or to another guide.

CONTENTS

INSTALLATION AND DOCUMENTATION ROUTE MAP

This chapter explains how to install and run the command line and Windows Workbench versions of the IAR products, and gives an overview of the user guides supplied with them.

Please note that some products only exist in a command line version, and that the information may differ slightly depending on the product or platform you are using.

COMMAND LINE VERSIONS

This section describes how to install and run the command line versions of the IAR Systems tools.

WHAT YOU NEED

- DOS 4.x or later. This product is also compatible with a DOS window running under Windows 95, Windows NT 3.51 or later, or Windows 3.1x.
- At least 10 Mbytes of free disk space.
- A minimum of 4 Mbytes of RAM available for the IAR applications.

INSTALLATION

1 Insert the first installation disk.

2 At the MS-DOS prompt type:

```
a:\install ↵
```

3 Follow the instructions on the screen.

When the installation is complete:

4 Make the following changes to your `autoexec.bat` file:

Add the paths to the IAR Systems executable and user interface files to the `PATH` variable; for example:

```
PATH=c:\dos;c:\utils;c:\iar\exe;c:\iar\ui;
```

Define environment variables `C_INCLUDE` and `XLINK_DFLTDIR` specifying the paths to the `inc` and `lib` directories; for example:

```
set C_INCLUDE=c:\iar\inc\
set XLINK_DFLTDIR=c:\iar\lib\
```

5 Reboot your computer for the changes to take effect.

6 Read the Read-Me file, named *product*`.doc`, for any information not included in the guides.

RUNNING THE TOOLS

Type the appropriate command at the MS-DOS prompt.

For more information refer to the chapter *Getting started* in the *Command Line Interface Guide*.

WINDOWS WORKBENCH VERSIONS

This section explains how to install and run the Embedded Workbench.

WHAT YOU NEED

- Windows 95, Windows NT 3.51 or later, or Windows 3.1x.
- Up to 15 Mbytes of free disk space for the Embedded Workbench.
- A minimum of 4 Mbytes of RAM for the IAR applications.

If you are using C-SPY you should install the Workbench before C-SPY.

INSTALLING FROM WINDOWS 95 OR NT 4.0

1 Insert the first installation disk.

2 Click the **Start** button in the taskbar, then click **Settings** and **Control Panel**.

3 Double-click the **Add/Remove Programs** icon in the **Control Panel** folder.

4 Click **Install**, then follow the instructions on the screen.

RUNNING FROM WINDOWS 95 OR NT 4.0

1 Click the **Start** button in the taskbar, then click **Programs** and **IAR Embedded Workbench**.

2 Click **IAR Embedded Workbench**.

INSTALLING FROM WINDOWS 3.1x OR NT 3.51

1. Insert the first installation disk.
2. Double-click the **File Manager** icon in the **Main** program group.
3. Click the **a** disk icon in the **File Manager** toolbar.
4. Double-click the **setup.exe** icon, then follow the instructions on the screen.

RUNNING FROM WINDOWS 3.1x OR NT 3.51

1. Go to the Program Manager and double-click the **IAR Embedded Workbench** icon.

RUNNING C-SPY

Either:

1. Start C-SPY in the same way as you start the Embedded Workbench (see above).

Or:

1. Choose **Debugger** from the Embedded Workbench **Project** menu.

UNIX VERSIONS

This section describes how to install and run the UNIX versions of the IAR Systems tools.

WHAT YOU NEED

- HP9000/700 workstation with HP-UX 9.x (minimum), or a Sun 4/SPARC workstation with SunOS 4.x (minimum) or Solaris 2.x (minimum).

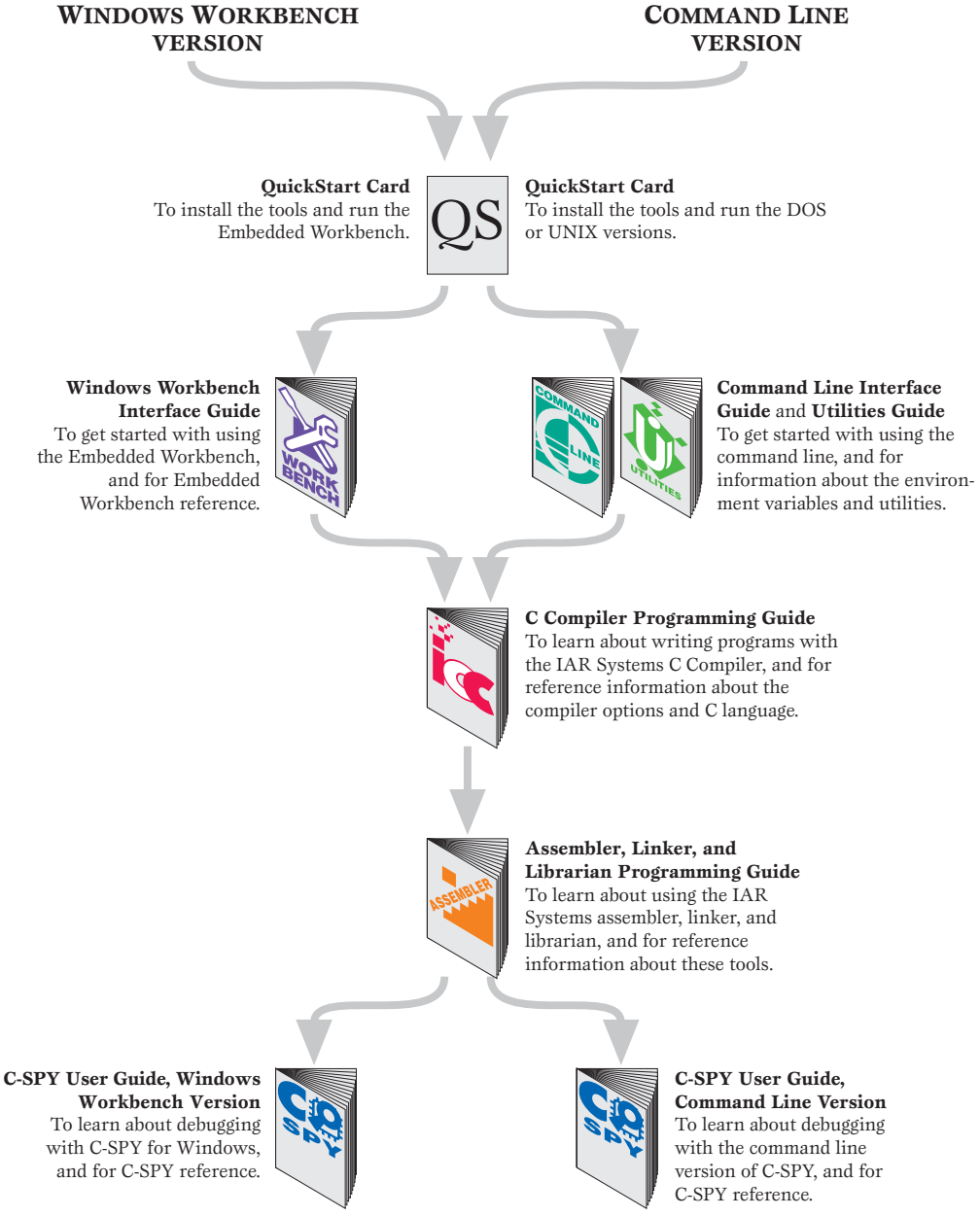
INSTALLATION

Follow the instructions provided with the media.

RUNNING THE TOOLS

Type the appropriate command at the UNIX prompt. For more information refer to the chapter *Getting started* in the *Command Line Interface Guide*.

DOCUMENTATION ROUTE MAP



INTRODUCTION

This guide describes the IAR Systems H8 Assembler, and its associated tools the XLINK Linker and XLIB Librarian, and provides information about running them from the command line.

ASSEMBLER

The IAR Systems H8 Assembler is a powerful relocating macro assembler with a versatile set of directives.

The assembler incorporates a high degree of compatibility with the microprocessor manufacturer's own assemblers, to ensure that software originally developed using them can be transferred to the IAR Systems Assembler with little or no modification.

It provides the following features:

GENERAL

- One pass assembly, for faster execution.
- Integration with the XLINK Linker and XLIB Librarian.
- Integration with other IAR Systems software.
- Self-explanatory error messages.

ASSEMBLER FEATURES

- Structured control directives.
- Support for H8/300H and H8S processors.
- Up to 256 relocatable segments per module.
- 32-bit arithmetic and IEEE floating-point constants.
- 255 significant characters in symbols.
- Powerful recursive macro facilities.
- Number of symbols and program size limited only by available memory.
- Support for complex expressions with external references.
- Forward references allowed to any depth.

- Support for C language pre-processor directives and `sfr` keyword.
- Macros in Intel/Motorola style.

XLINK LINKER

The IAR Systems XLINK Linker converts one or more relocatable object files produced by the IAR Systems Assembler or C Compiler to machine code for a specified target processor. It supports a wide range of industry-standard loader formats, in addition to the IAR Systems debug format used by the C-SPY high level debugger.

XLINK supports user libraries, and will load only those modules that are actually needed by the program you are linking.

The final output produced by XLINK is an absolute, target-executable object file that can be programmed into an EPROM, downloaded to a hardware emulator, or run directly on the host using the IAR Systems C-SPY debugger.

XLINK offers the following important features:

FEATURES OF XLINK

- Unlimited number of input files.
- Searches user-defined library files and loads only those modules needed by the application.
- Symbols may be up to 255 characters long with all characters being significant. Both upper and lower case may be used.
- Global symbols can be defined at link time.
- Flexible segment commands allow full control of the locations of relocatable code and data in memory.
- Support for over 30 emulator formats.

XLIB LIBRARIAN

The IAR Systems XLIB Librarian enables you to manipulate the relocatable object files produced by the IAR Systems Assembler and C Compiler.

XLIB provides the following features:

FEATURES OF XLIB

- Support for modular programming.
- Modules can be listed, added, inserted, replaced, deleted, or renamed.
- Segments can be listed and renamed.
- Symbols can be listed and renamed.
- Modules can be changed between program and library type.
- Interactive or batch mode operation.
- A full set of library listing operations.
- On-line help.

TUTORIAL

This tutorial illustrates how you might use the H8 Assembler to develop a series of simple machine-code programs for the H8 processor, and illustrates some of the assembler's most important features.

Before reading this chapter you should:

- Have installed the assembler software; see the *QuickStart Card* or the chapter *Installation and documentation route map*.
- Be familiar with the architecture and instruction set of the H8 processor. For more information see the chapter *Assembler instructions*, and the manufacturer's data book.

It is also recommended that you complete the introductory tutorial in the *H8 Command Line Interface Guide*, to familiarize yourself with the interface you are using.

RUNNING THE EXAMPLE PROGRAMS

This tutorial shows how to run the example programs using the optional C-SPY debugger.

Alternatively, you can run the examples by linking them without debugging information to give a file `aout.a37`, which can be downloaded to an emulator with debugging facilities. Use the XLINK `-F` option to specify a format other than the default, Motorola extended.

GETTING STARTED

The first step is to create a new project for the tutorial programs.

CREATING A NEW PROJECT

It is a good idea to keep all the files for a particular project in one directory, separate from other projects and the system files.

The tutorial files are installed in the `ah8` directory. Select this directory by entering the command:

```
cd c:\iar\ah8 ↵
```

During this tutorial, you will work in this directory, so that the files you create will reside here.

CREATING A PROGRAM

The first tutorial illustrates how you write a basic assembler program, and how you then assemble, link, and run it.

WRITING A PROGRAM

The first example program is a simple count loop which counts up the R0 register in binary-coded decimal:

```
        NAME    first

        ORG     0       ; Vector table 0 - FF
        DC.W    main    ; Reset vector

; Counts up R0 in binary-coded decimal

        ORG     100
main    MOV     #0,R0
loop    ADD.B   #1,R0L
        DAA     R0L
        ADDX.B  #0,R0H
        DAA     R0H
        BNE     loop
        CMP     #0,R0L
        BNE     loop
        RTS

        END     main
```

The `ORG` directive assembles the program starting at address `100h`, and this address is assembled in the reset vector so the program is executed upon reset.

Enter the program using any standard text editor, such as the MS-DOS `edit` editor, and save it in a file called `first.s37`. The files associated with the H8 Assembler have extensions `.s37`, `.a37`, `.d37`, and `.r37` to identify them. Alternatively, a copy is provided in the assembler files directory.

You now have a source file which is ready to assemble.

ASSEMBLING THE PROGRAM

To assemble the file, type the following command at the prompt:

```
ah8 first -v0 -r -L ↵
```

The `-v0` option assembles code for the H8/300H and the `-r` option includes debugging information for use by C-SPY. The `-L` option sends a listing to the file `first.lst`.

Viewing the listing

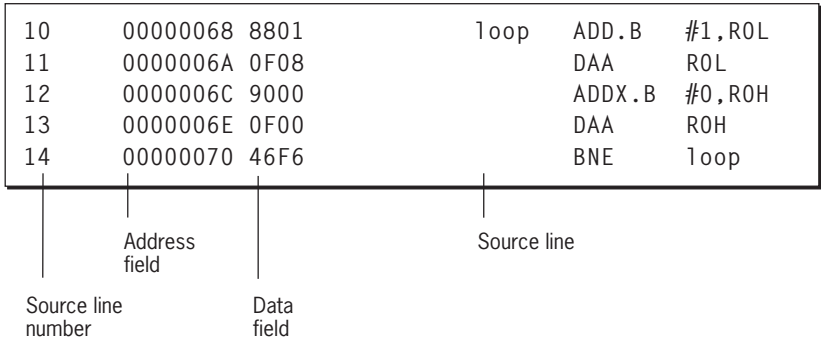
If you look at the listing file you will see that it contains the following:

```
##############################################################################
#   IAR Systems H8 Assembler Vx.xx                                           #
#                                                                            #
#           Target option =  H8/300H                                         #
#           Source file   =  first.s37                                       #
#           List file     =  first.lst                                       #
#           Object file   =  first.r37                                       #
#           Command line  =  first -v0 -r -L                                 #
#                                                                            #
#                                        (c) Copyright IAR Systems 1996      #
##############################################################################
      1    00000000                   NAME    first
      2    00000000
      3    00000000                   ORG     0       ; Vector table 0 - FF
      4    00000000 0064              DC.W    main    ; Reset vector
      5    00000002
      6    00000002          ; Counts up R0 in binary-coded decimal
      7    00000002
      8    00000064                   ORG     100
      9    00000064 79000000 main     MOV     #0,R0
     10    00000068 8801     loop     ADD.B   #1,R0L
     11    0000006A 0F08              DAA     R0L
     12    0000006C 9000              ADDX.B  #0,R0H
     13    0000006E 0F00              DAA     R0H
     14    00000070 46F6              BNE     loop
     15    00000072 A800              CMP     #0,R0L
     16    00000074 46F2              BNE     loop
     17    00000076 5470              RTS
     18    00000078
     19    00000078                   END     main
##############################
#          CRC:1576          #
#        Errors:   0         #
#        Warnings: 0         #
#         Bytes: 22          #
##############################
```

This shows the machine-code instructions generated by each of the source code statements.

Note that the CRC number depends on the date of assembly, and may vary.

The format of the listing is as follows:



Assuming that the source assembled successfully, a further file, `first.r37`, will also be created, containing the linkable object code.

If you made any errors when entering the program, these will be displayed on the screen during the assembly. If this happens, return to the editor, check carefully through the source code to locate and correct all the mistakes, resave the source file using the same name, and try assembling it again.

LINKING THE PROGRAM

To link the object file to produce code that can be executed, enter the command:

```
xlink first -ch8 -r ↵
```

The `-c` option specifies H8/300H as the target processor, and the `-r` option includes debugging information.

In this simple example there is only one segment, the default segment, which will be located in memory starting at address 0.

By default, the output code will be placed in a file `aout.d37`.

RUNNING THE PROGRAM

To run the example program using the C-SPY emulator give the command:

```
csh8 aout -v0 ↵
```

In C-SPY give the commands:

```
WINDOW REG ON ↵
```

and then press F2 to step through the program and watch R0 count in BCD.

USING MACROS

The second sample program will be used to demonstrate the use of a simple macro. It defines a `DayofWeek` subroutine which calculates the day of the week for any day in the twentieth century.

The logic of the program is as follows:

```
int date, month, year;
if (month>2)
  month = month-2;
else
  {year = year-1; month = month+10};
x = ((26*month-2) div 10) + date + year + (year div 4) + 1;
printf("Day of week: %d", x mod 7);
```

The program defines the following two macros, to divide and multiply R1 by an 8-bit constant, using R3L as a temporary register:

```
divi    MACRO   q
        MOV     #q,R3L
        DIVXU   R3L,R1
        ENDM

muli    MACRO   q
        MOV     #q,R3L
        MULXU   R3L,R1
        ENDM
```

The `divi` macro has a parameter, `q`, which substitutes its argument in the `MOV` instruction. It is called with a statement such as:

```
        divi    7
```

This assembles into the instructions:

```
        MOV     #7,R3L
        DIVXV   R3L,R1
```

The full listing of the DayofWeek assembler program is as follows:

```
        NAME    DayofWeek

        ORG     0
        DC.W    daydate

        RSEG    DATA
day     DS.B    1

        RSEG    CODE
;
;       DayDate subroutine
;       On Entry        R0L = Year (0..99)
;                       R1L = Month (1..12)
;                       R2L = Date (0..31)
;
;       On Exit  day = Day of the week
;
divi    MACRO   q
        MOV     #q,R3L
        DIVXU   R3L,R1
        ENDM

muli    MACRO   q
        MOV     #q,R3L
        MULXU   R3L,R1
        ENDM

daydate CMP     #2,R1L
        BLS     janfeb
        SUBX    #2,R1L
        BRA     cont
janfeb  DEC     R0L
        ADD     #10,R1L
cont    muli    26
        SUBX    #2,R1L
        divi    10
```

```
        MOV     #0,R1H
        ADD     R2L,R1L
        ADD     R0L,R1L
        SHLR    R0L
        SHLR    R0L
        ADD     R0L,R1L
        INC     R1L
        divi    7
        MOV     R1H,@day
        RTS

        END     daydate
```

The program assumes that the year, month, and date are in the R0L, R1L, and R2L registers, respectively, and it returns the day of the week in the memory location `day`, with 0 = Sunday, 1 = Monday … 6 = Saturday, etc.

Type in this listing and save it in a file `day.s37`. Alternatively, a copy of the source is provided on the installation disk.

ASSEMBLING THE PROGRAM

To assemble the source program enter the command:

```
ah8 day -v0 -r -L ↵
```

Viewing the listing

The following output will be produced in the file `day.lst`. In this and subsequent listings the header information is omitted for clarity:

```
 1    00000000                    NAME    DayofWeek
 2    00000000
 3    00000000
 4    00000000                    ORG     0
 5    00000000 ....               DC.W    daydate
 6    00000002
 7    00000000                    RSEG    DATA
 8    00000000           day      DS.B    1
 9    00000001
10    00000000                    RSEG    CODE
11    00000000           ;
12    00000000           ;   DayDate subroutine
13    00000000           ;   On Entry R0L = Year (0..99)
```

```
14    00000000          ;            R1L = Month (1..12)
15    00000000          ;            R2L = Date (0..31)
16    00000000          ;
17    00000000          ;   On Exit  day = Day of the week
18    00000000          ;
23    00000000
28    00000000
29    00000000 A902     daydate CMP     #2,R1L
30    00000002 4304             BLS     janfeb
31    00000004 B902             SUBX    #2,R1L
32    00000006 4004             BRA     cont
33    00000008 1A08     janfeb  DEC     R0L
34    0000000A 890A             ADD     #10,R1L
35    0000000C          cont    muli    26
35.1  0000000C FB1A             MOV     #26,R3L
35.2  0000000E 50B1             MULXU   R3L,R1
35.3  00000010                  ENDM
36    00000010 B902             SUBX    #2,R1L
37    00000012                  divi    10
37.1  00000012 FB0A             MOV     #10,R3L
37.2  00000014 51B1             DIVXU   R3L,R1
37.3  00000016                  ENDM
38    00000016 F100             MOV     #0,R1H
39    00000018 08A9             ADD     R2L,R1L
40    0000001A 0889             ADD     R0L,R1L
41    0000001C 1108             SHLR    R0L
42    0000001E 1108             SHLR    R0L
43    00000020 0889             ADD     R0L,R1L
44    00000022 0A09             INC     R1L
45    00000024                  divi    7
45.1  00000024 FB07             MOV     #7,R3L
45.2  00000026 51B1             DIVXU   R3L,R1
45.3  00000028                  ENDM
46    00000028 6AA100..         MOV     R1H,@day
47    0000002E 5470             RTS
48    00000030
49    00000030                  END     daydate
```

The macro-generated lines are numbered with a decimal suffix: eg 35.1, 35.2, etc.

The address for the location `day` is show as '.' in the `MOV` instruction on line 46, and will be resolved when the program is linked.

LINKING THE PROGRAM

In order to be able to execute the program, the relocatable file produced by the assembler needs to be converted to an object code program with all the addresses resolved.

Run XLINK to produce code for debugging with the command:

```
xlink day -ch8 -r -l day.map -ZCODE,DATA=100 ↵
```

This generates a file `aout.d37`.

This time the module has two segments, `DATA` and `CODE`, and we use the linker `-Z` option to specify that `DATA` should be placed immediately after `CODE` in memory, starting at address `100` to leave space for the vector table between `0` and `FF`.

RUNNING THE PROGRAM

If you have the C-SPY simulator you can run the program with the command:

```
csh8 aout -v0 ↵
```

Set a breakpoint at the `RTS` instruction at the end of the program, by moving the source cursor to the `RTS` line with Ctrl N, and then typing:

```
BREAK SET ↵
```

Then set up the registers using the `REGISTER` command; for example, to find the day of the week for the 29th January 1955 type:

```
REG R0L = 55t ↵
REG R1L = 1 ↵
REG R2L = 29t ↵
```

Then run the program by typing:

```
GO ↵
```

The result will be put in the memory location `day`.

GENERATING A FILE FOR A PROM PROGRAMMER

To generate code which can be read by a PROM programmer, link without the `-r` option to get a file `aout.a37`.

USING STRUCTURED ASSEMBLY

This example demonstrates how to take full advantage of the H8 Assembler's powerful structured assembly facilities.

In this example a simple routine is first written in conventional assembler instructions. The example then shows how it can be rewritten using structured assembly directives to reduce the number of source lines, make the program logic clearer, and reduce the possibility of programmer error.

WRITE THE PROGRAM

The following program implements Euclid's algorithm for finding the greatest common divisor of two numbers in the registers R0L and R1L:

```
/* Euclids's Algorithm */
/* Numbers in R0L and R1L */
/* Result is in R1L */

        NAME    GCD

        RSEG    CODE
begin   MOV     #0,R1H
        DIVXU   R0L,R1  ;Remainder = R1H
        MOV     R1H,R1L
        CMP     R1L,R0L
        BLE     noswap
        MOV     R0L,R1L
        MOV     R1H,R0L
noswap  CMP     #0,R0L
        BNE     begin
        RTS     ; Result in R1L

        END     begin
```

Type the program into a file `gcd1.s37`. Alternatively, a copy is supplied on the installation disk.

ASSEMBLE THE PROGRAM

Assemble the `gcd1` program with the following command:

```
ah8 gcd1 -v0 -r -L ↵
```

The listing will be sent to the file `gcd.lst`. This is as follows:

```
 1  00000000      /* Euclids's Algorithm */
 2  00000000      /* Numbers in R0L and R1L */
 3  00000000      /* Result is in R1L */
 4  00000000
 5  00000000              NAME    GCD
 6  00000000
 7  00000000              RSEG    CODE
 8  00000000
 9  00000000 F100   begin MOV     #0,R1H
10  00000002 5181         DIVXU   R0L,R1 ;Remainder = R1H
11  00000004 0C19         MOV     R1H,R1L
12  00000006 1C98         CMP     R1L,R0L
13  00000008 4F04         BLE     noswap
14  0000000A 0C89         MOV     R0L,R1L
15  0000000C 0C18         MOV     R1H,R0L
16  0000000E A800   noswap CMP    #0,R0L
17  00000010 46EE         BNE     begin
18  00000012 5470         RTS     ; Result in R1L
19  00000014
20  00000014              END     begin
```

USING STRUCTURES

The IAR Assembler provides the following directives for structured assembly:

Construct	*Description*
`IFS … ELSEIFS … ELSES … ENDIFS`	Specifies instructions to be executed if a condition is true.
`WHILE … ENDW`	Executes a loop as long as an expression is true.
`REPEAT … UNTIL`	Executes a loop until an expression is true.
`FOR … ENDF`	Repeats subsequent instructions a specified number of times.
`SWITCH … ENDS`	Multiple case switch.

The gcd program can be rewritten to use the REPEAT … UNTIL loop, and an IFS … ENDIFS test, as follows:

```
/* Euclids's Algorithm */
/* Numbers in R0L and R1L */
/* Result is in R1L */

        NAME    GCD

        RSEG    CODE

begin   REPEAT
        MOV     #0,R1H
        DIVXU   R0L,R1  ;Remainder = R1H
        MOV     R1H,R1L
        IFS     R0L <GT> R1L THEN
        MOV     R0L,R1L
        MOV     R1H,R0L
        ENDIFS
        UNTIL   R0L <EQ> #0
        RTS     ; Result in R1L

        END     begin
```

Save this in a file gcd2.s37.

Assemble this with the command:

```
ah8 gcd2 -v0 -r -L ↵
```

This produces the following code:

```
  1    00000000       /* Euclids's Algorithm */
  2    00000000       /* Numbers in R0L and R1L */
  3    00000000       /* Result is in R1L */
  4    00000000
  5    00000000             NAME    GCD
  6    00000000
  7    00000000             RSEG    CODE
  8    00000000
  9    00000000      begin  REPEAT
  9.1  00000000       _?0
 10    00000000 F100        MOV     #0,R1H
 11    00000002 5181        DIVXU   R0L,R1  ;Remainder = R1H
```

```
12    00000004 0C19        MOV     R1H,R1L
13    00000006             IFS     R0L <GT> R1L THEN
13.1  00000006 1C98        CMP R1L,R0L
13.2  00000008 4F04        BLE _?2
14    0000000A 0C89        MOV     R0L,R1L
15    0000000C 0C18        MOV     R1H,R0L
16    0000000E             ENDIFS
16.1  0000000E       _?2
17    0000000E             UNTIL   R0L <EQ> #0
17.1  0000000E A800        CMP #0,R0L
17.2  00000010 46EE        BNE _?0
17.3  00000012       _?1
18    00000012 5470        RTS     ; Result in R1L
19    00000014
20    00000014             END     begin
```

The code generated by the structured assembly directives is listed on lines with a decimal suffix: `13.1`, `13.2`, etc. The assembler generates extra labels `_?0`, `_?1`, etc to implement the generated branch instructions.

The main `REPEAT` … `UNTIL` loop produces similar code for the loop as in the original example but in a much more convenient and easy to read manner.

Similarly the test in the middle of the body of the loop has been replaced with an `IFS` … `ENDIFS` structure. This tests the register R0L against register R1L. The `<GT>` test results in a `CMP` instruction followed by a `BLE` around the body of the section (which swaps the registers R0L and R1L).

USING MODULES

The final example demonstrates how to create library modules and use the XLIB Librarian to maintain files of modules.

USING LIBRARIES

If you are working on a large project you will soon accumulate a collection of useful routines that are used by several of your programs.

To avoid the need to assemble a routine each time you need it you can store such routines as object files; ie assembled but not linked.

A collection of routines in a single object file is referred to as a library. It is recommended that you use library files to create collections of related routines, such as graphical or math libraries.

You can use the XLIB Librarian to manipulate libraries; it allows you to:

- Change modules from PROGRAM to LIBRARY type, and vice versa.
- Add or remove modules from a library file.
- Change the names of entries.
- List module names, entry names, etc.

CREATING THE MAIN PROGRAM

The main program is as follows:

```
        NAME    main

        PUBLIC  main
        EXTERN  rightshift

        RSEG    PROM
main    MOV     #H'ABCD,R0
        MOV     #4,R1L
        JSR     rightshift
        SLEEP

        END     main
```

This simply uses a routine called `rightshift` to shift the contents of register R0 to the right. The data in register R0 is set to `H'ABCD` and the `rightshift` routine is called to shift it to the right by four places as specified by the contents of register R1.

The `EXTERN` directive declares `rightshift` as an external symbol, to be resolved at link time.

Enter this program and save it as the file `main.s37` or, alternatively, copy the file provided in the assembler files directory (by default `c:\iar\ah8`).

CREATING THE LIBRARY ROUTINES

The second program is used to form a separately assembled library. This contains two library routines: the `rightshift` routine called by `main`, and the corresponding `leftshift` routine. These both operate on the contents of register R0 by repeatedly shifting it to the right or left. The number of shifts performed is controlled by decrementing register R1L to zero.

```
        MODULE  rightshift
        PUBLIC  rightshift
        RSEG    PROM
rightshift
        WHILE   R1L <GT> #0 DO
        SHLR    R0H
        ROTXR   R0L
        DEC     R1L
        ENDW
        RTS
        ENDMOD

        MODULE  leftshift
        PUBLIC  leftshift
        RSEG    PROM
leftshift
        WHILE   R1L <GT> #0 DO
        SHLL    R0L
        ROTXL   R0H
        DEC     R1L
        ENDW
        RTS

        END
```

The routines are defined as library modules by the `MODULE` directives; these instruct the XLINK Linker to include them only if they are called by another module.

The `rightshift` and `leftshift` entry addresses are made public to other modules with a `PUBLIC` directive.

Save these modules in a source file called `shifts.s37` or, alternatively, copy the file provided in the assembler files directory (by default `c:\iar\ah8`).

ASSEMBLING AND LINKING THE SOURCE FILES

Next you need to assemble both of the above source files.

Although it is possible to assemble both source files together, in a large project this would soon become very time-consuming. By assembling the library routines separately, changes to the main program only require reassembly of the main source file.

To assemble the main program type:

```
ah8 main -v0 ↵
```

Similarly, to assemble the library routines type:

```
ah8 shifts -v0 ↵
```

Assembling the files creates two relocatable files. You need to link these together to produce a single executable object file containing the main program and the library routine it references, with all of the cross references resolved. In this case the only reference from one section to the other is the call of the `max` subroutine. The `min` routine is not used at all.

To link the files in a single step enter the following at the command line (on one line):

```
xlink main shifts -ch8 -ZPROM=4000 -xsm -l main.map ↵
```

The following table explains the options which define the addresses for the code and data segments:

Parameter	*Description*
`-ZPROM=4000`	Defines that the code segment is to be relocated to the hex address 4000.
`-xsm`	Requests a cross reference listing.
`-l main.map`	Directs the listing output to `main.map`.

For more information about the XLINK options see the chapter *XLINK options reference*.

Viewing the listing

If you list the cross reference listing, `main.map`, you will see that the module created by XLINK includes the `main` program module and the `rightshift` library module, but not the unused `leftshift` library module.

USING THE XLIB LIBRARIAN

Once you have assembled and debugged a module intended for general use, like the `max` and `min` modules previously described, you can add them to a library using the XLIB Librarian.

Running the XLIB Librarian

Start the XLIB Librarian by typing:

```
XLIB ↵
```

XLIB runs in an interactive mode, and displays a * prompt for you to enter your command.

The first thing you need to do within XLIB is define the CPU you are using:

```
DEFINE-CPU h8 ↵
```

Giving XLIB commands

Extract the modules you want from `maxmin.r37` into a library called `math.r37`. To do this enter the command:

```
FETCH-MODULES ↵
```

This prompts for the following arguments:

Prompt	*What you type*
Source file	`shifts` ↵
Destination file	`math` ↵
Start module	↵ (uses the default, which is the first in the file).
End module	↵ (uses the default, which is the last in the file).

This creates the file `math.r37` which contains the code for the `leftshift` and `rightshift` routines.

You can confirm this by typing:

```
LIST-MODULES ↵
```

This prompts for the following arguments:

Prompt	*What you type*
Object file	`math`
List file	↵ (to use the screen).
Start module	↵ (to start from the first module).
End module	↵ (to end at the last module).

Finally, leave the librarian by typing:

```
EXIT ↵
```

You could use the same procedure to add further modules to the `math` library at any time.

ASSEMBLER OPTIONS SUMMARY

This chapter explains how to set the assembler options from the command line.

The options are divided into the following sections:

Code generation	#undef
Debug	Include
#define	Target
List	Miscellaneous
Macro	

For full reference about each option refer to the following chapter, *Assembler options reference*.

SETTING ASSEMBLER OPTIONS

To set assembler options from the command line, you include them on the command line, after the `ah8` command. For example, when assembling the source `power2`, to generate a listing to the default listing filename (`power2.lst`):

```
ah8 power2 -L ↵
```

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

```
ah8 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 listing to the default filename but in the subdirectory list:

```
ah8 power2 -Llist ↵
```

Options can also be specified in the `ASMH8` environment variable. The assembler automatically 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.

For example, putting the following line in the `autoexec.bat` file:

```
set ASMH8=-l temp.lst
```

will always generate a listing to `temp.lst`.

OPTIONS SUMMARY

The following is a summary of all the assembler options. For a full description of any option, see under the option's category name in the next chapter, *Assembler options reference*.

Option	*Description*	*Section*
`-B`	Macro execution info.	List
`-b`	Make object a library module.	Miscellaneous
`-c{ACDEMOS}`	Conditional list.	List
`-D`*`symb`*`[=`*`xx`*`]`	Define symbol.	#define
`-d`	Disable `#ifdef` check.	#define
`-E`*`number`*	Max number of errors.	Miscellaneous
`-f` *`filename`*	Extend the command line.	Miscellaneous
`-G`	Open standard input as source.	Miscellaneous
`-I`*`prefix`*	Include paths.	Include
`-i`	Add `#include` file.	List
`-L[`*`prefix`*`]`	List to prefixed source name.	List
`-l` *`filename`*	List to named file.	List
`-M`*`ab`*	Macro quote chars.	Macro
`-m{s|l}`	Memory model.	Target
`-N`	No header.	List
`-O`*`prefix`*	Set object filename prefix.	Miscellaneous
`-o` *`filename`*	Set object filename.	Miscellaneous
`-p`*`lines`*	Lines/page.	List
`-r`	Generate debug information.	Debug
`-S`	Set silent operation.	Miscellaneous

Option	*Description*	*Section*
`-s{+|-}`	User symbols case sensitivity.	Code generation
`-T`	List active lines only.	List
`-t`*`n`*	Tab spacing.	List
`-U`*`symb`*	Undefine symbol.	#undef
`-v`*`n`*	Chip option	Target
`-w[`*`string`*`]`	Disable warnings.	Code generation
`-x{DI2}`	Cross-reference.	List

ASSEMBLER OPTIONS REFERENCE

This chapter gives detailed information on each of the H8 Assembler options, divided into functional categories.

CODE GENERATION

These options control the assembler's code generation.

`-s{+|-}` User symbols case sensitivity.

`-w[`*`string`*`]` Disable warnings.

USER SYMBOLS CASE SENSITIVITY (-s)

Syntax: `-s{+|-}`

Sets whether the assembler is sensitive to the case of user symbols:

Option	*Command line*
Case insensitive user symbols	`-s-` (default)
Case sensitive user symbols	`-s+`

By default, case sensitivity is off. This means that, for example, `LABEL` and `label` refer to the same symbol. You can choose **Case sensitive user symbols** (`-s+`) to turn case sensitivity on, in which case `LABEL` and `label` will refer to different symbols.

DISABLE WARNINGS (-w)

Syntax: `-w[`*`string`*`]`

Disables warnings.

By default, the assembler displays a warning message when it finds an element of the source which might be erroneous, due to a programming error (see *Assembler diagnostics* for details). The **Disable warnings** (`-w`) option with no range disables all warnings. The **Disable warnings** (`-w`) option with a range performs the following:

Range	*Effect*
+	Enables all warnings.
-	Disables all warnings.
+*n*	Enables just warning *n*.
-*n*	Disables just warning *n*.
+*m*-*n*	Enables warnings *m* to *n*.
-*m*-*n*	Disables warnings *m* to *n*.

For example, to disable just warning 0 (unreferenced label), you might use:

```
ah8 prog -w-0 ↵
```

or to disable warnings 0 to 8:

```
ah8 prog -w-0-8 ↵
```

Only one **Disable warnings** (`-w`) option may be used on the command line.

DEBUG

The **Debug** option provides information for debugging with C-SPY.

`-r` Generate debug information.

GENERATE DEBUG INFORMATION (-r)

Syntax: `-r`

Enables the inclusion of information that allows a 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 **Generate debug information** (`-r`) option if you want to use a debugger with the program.

Note that `-r` will not provide source level debugging in C-SPY. However, labels within typed segments will be shown; see *Define segments (-Z)*, page 180. Defined symbols can be listed with the C-SPY `SYMBOLS` command.

#define

This option allows you to define symbols.

`-D`*`symb`*`[=`*`xx`*`]` Define symbol.

`-d` Disable `#ifdef` check.

DEFINE SYMBOL (-D)

Syntax: `-D`*`symb`*`[=`*`xx`*`]`

Defines a symbol with the name *`symb`* and the value *`xx`*. If no value is specified, 1 is used.

The **Define symbol** (`-D`) option allows a value or choice that would otherwise have to be specified in the source file to be specified more conveniently on the command line. 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 you would use include sections such as:

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

Then, you would select the version required in the command line as follows:

```
production version:     ah8 prog
test version:           ah8 prog -Dtestver
```

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

```
ah8 prog -Dframerate=3 ↵
```

DISABLE #IFDEF CHECK (-d)

Syntax: `-d`

`#ifdef` is checked against `#else` and `#endif` at the end of a module. You can use the `-d` command line option to disable the test. This will then allow programs like:

```
#define FOO
#ifdef  FOO
        module m1
```

```
        nop
        endmod
#endif
        module m2
        nop
        end
```

LIST

The **List** options are used to cause the assembler to generate a listing, to select the contents of the listing, and to generate other listing-type output.

`-B`	Macro execution info.
`-c{ACDEMOS}`	Conditional list.
`-i`	Add `#include` file.
`-L[`*`prefix`*`]`	List to prefixed source name.
`-l` *`filename`*	List to named file.
`-N`	No header.
`-p`*`lines`*	Lines/page.
`-T`	List active lines only.
`-t`*`n`*	Tab spacing.
`-x{DI2}`	Cross-reference.

MACRO EXECUTION INFO (-B)

Syntax: `-B`

Causes the assembler to 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.

CONDITIONAL LIST (-c)

Syntax: `-c{ACDEMOS}`

Set one or more of the following:

Option	*Command line*
Assembled part only	`A`
List total cycle count	`C`
Disable list	`D`
No macro expansion list	`E`
List macro definition	`M`
List multiline code	`O`
No structured assembly expansion list	`S`

ADD #INCLUDE FILE (-i)

Syntax: `-i`

Causes `#include` files to be included in the listing.

By default, the assembler does not list `#include` file lines since these are often from standard files that would waste space in the listing. The **Add #include file** (`-i`) option allows you to list `#include` files should you so require.

LIST TO PREFIXED SOURCE NAME (-L)

Syntax: `-L[`*`prefix`*`]`

Causes the assembler to generate a listing and send it to the file *`prefixsourcename`*`.lst`. Note that you must not include a space before the prefix.

By default, the assembler does not generate a listing. To simply generate a listing, you use the `-L` option without a prefix. The listing is sent to the file with the same name as the source, but extension `.lst`.

The `-L` option lets you specify a prefix, for example to direct the list file to a subdirectory:

```
ah8 prog -Llist\ ↵
```

This sends the object to `list\prog.lst` rather than the default `prog.lst`.

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

LIST TO NAMED FILE (-l)

Syntax: `-l` *`filename`*

Causes the assembler to generate a listing and send it to the named file. If no extension is specified, `.lst` is used. Note that you must include a space before the filename.

By default, the assembler does not generate a listing. The `-l` option turns on listing, and directs it to a specific file. To just turn on listing to the default filename, use the `-L` option instead.

NO HEADER (-N)

Syntax: `-N`

Disables the header normally printed in the listing.

LINES/PAGE (-p)

Syntax: `-p`*`lines`*

Sets the number of lines per page to *`lines`*, which must be in the range 10 to 150.

LIST ACTIVE LINES ONLY (-T)

Syntax: `-T`

Causes listing to include only active lines, for example not those in false `#if` blocks. By default, all lines are listed.

This option is useful for reducing the size of listings by eliminating lines that do not generate or affect code.

TAB SPACING (-t)

Syntax: `-t`*n*

Set the number of character positions per tab stop to *n*, which must be in the range 2 to 9.

By default, the assembler sets eight character positions per tab stop.

CROSS-REFERENCE (-x)

Syntax: `-x{DI2}`

Causes the assembler to generate a cross-reference list at the end of the listing. See the chapter *Assembler file formats* for details.

The following options are available:

Option	*Command line*
Show #defines	`D`
Show internal symbols	`I`
Dual line spacing	`2`

MACRO

The **Macro** option controls the interpretation of macros.

`-M`*ab* Macro quote chars.

MACRO QUOTE CHARS (-M)

Syntax: `-M`*ab*

Sets the characters used for the left and right quotes of each macro argument to `a` and `b` respectively.

By default, the characters are < and >. The **Macro quote chars** (`-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.

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

to call a macro print with > as the argument.

#undef

The **#undef** option allows you to undefine predefined symbols.

`-U`*`symb`* Undefine symbol.

UNDEFINE SYMBOL (-U)

Syntax: `-U`*`symb`*

Undefines the symbol *`symb`*.

By default, the assembler provides certain pre-defined symbols; see *Pre-defined symbols*, page 48. The **Undefine symbol** (`-U`) option allows you to undefine such a pre-defined symbol to make its name available for your own use through a subsequent **Define symbol** (`-D`) option or source definition.

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

```
ah8 prog -U __TIME__ ↵
```

INCLUDE

The **Include** option allows you to define the include path for the assembler.

`-I`*`prefix`* Include paths.

INCLUDE PATHS (-I)

Syntax: `-I`*`prefix`*

Adds the `#include` file search prefix *`prefix`*.

By default, the assembler searches for `#include` files only in the current working directory. The **Include paths** (`-I`) option allows you to give the assembler the names of directories which it will also search if it fails to find the file in the current working directory.

For example, using the options:

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

and then writing:

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

in the source, will make the assembler search first for file `asmlib.hdr`, then for file `c:\global\asmlib.hdr`, and finally for file `c:\thisproj\headers\asmlib.hdr`.

TARGET

The **Target** options specify the processor and memory model for the assembler and C compiler.

`-m{s|l}`	Memory model.
`-v`*n*	Chip option.

MEMORY MODEL (-m)

Syntax: `-m{s|l}`

Selects the memory model from the following options:

Option	*Description*
`-ms`	Small mode, 16-bit address space (default).
`-ml`	Large mode, 24- or 32-bit address space depending on the processor chosen.

CHIP OPTION (-v)

Syntax: `-v`*n*

Selects the processor version from one of:

Option	*Processor*
`-v0`	H8/300H (default)
`-v1`	H8S/2200
`-v2`	H8S/2600

If no **Chip option** (`-v`) option is specified, the assembler uses `-v0` by default.

MISCELLANEOUS

The following additional options are available.

`-b`	Make object a library module.
`-E`*`number`*	Max number of errors.
`-f` *`filename`*	Extend the command line.
`-G`	Open standard input as source.
`-O`*`prefix`*	Set object filename prefix.
`-o` *`filename`*	Set object filename.
`-S`	Set silent operation.

MAKE OBJECT A LIBRARY MODULE (-b)

Syntax: `-b`

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 XLIB. You use the `-b` option if you want it to make a library module for use with XLIB.

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

MAX NUMBER OF ERRORS (-E)

Syntax: `-E`*`number`*

Sets the maximum number of errors the assembler reports.

By default, the maximum number is 100. The **Max number of errors** (`-E`) option allows you to decrease or increase this number, for example, to see more errors in a single assembly.

EXTEND THE COMMAND LINE (-f)

Syntax: `-f filename`

Extends the command line with text read from the file *filename*`.xcl`. Note that there must be a space between the option itself and the filename.

The `-f` option is particularly useful where there are 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 `asmopt.xcl`, you might use:

```
ah8 prog -f asmopt ↵
```

OPEN STANDARD INPUT AS SOURCE (-G)

Syntax: `-G`

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

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

SET OBJECT FILENAME PREFIX (-O)

Syntax: `-O`*prefix*

Set the prefix to be used on the filename of the object. Note 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:

```
ah8 prog -Oobj\ ↵
```

This sends the object to `obj\prog.r37` rather than the default `prog.r37`.

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

SET OBJECT FILENAME (-o)

Syntax: `-o filename`

Sets the filename to be used for the object. Note that you must include a space before the filename. If no extension is specified, `.r37` is used.

By default the assembler uses the source filename with the extension changed to `.r37`. The `-o` option lets you use an alternative filename for the object.

For example, the following command puts the object to the file `obj.r37` instead of the default `prog.r37`:

```
ah8 prog -o obj ↵
```

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

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

SET SILENT OPERATION (-S)

Syntax: `-S`

Causes the assembler to operate without sending any messages to the standard output stream.

By default, the assembler sends various inessential messages to the terminal via the standard output stream. You can use the `-S` option to prevent this, reducing the amount of screen clutter. The assembler sends error and warning messages to the error output stream, so they appear on the terminal regardless.

ASSEMBLER FILE FORMATS

This chapter describes the source format for the H8 Assembler, and the format of assembler listings.

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 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`	One or more operands, separated by commas.
`comment`	A comment, preceded by a `;` (semi-colon).

The fields can be separated by spaces or tabs.

A source line may not exceed 255 characters.

Tab characters (ASCII 09H), are expanded according to the most common practice; ie to columns 8, 16, 24 etc.

EXPRESSIONS AND OPERATORS

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 to generate code.

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators.

The valid operands in an expression are:

- User-defined symbols and labels.

- Constants, excluding floating point constants.
- The location counter (PLC) symbol, `$`.

These are described in greater detail in the following sections.

The valid operators are described in the chapters *Assembler operator summary*, and *Assembler operator reference*.

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 where the segments are located by XLINK.

Such expressions are evaluated and resolved at link time, by XLINK. 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    TEST
        ORG     0
        DC.W    start

        RSEG    DATA
first   DS.B    5
second  DS.B    3
        ENDMOD

        RSEG    CODE
start   ...
```

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

```
        ADD.B   #first+7,R0L
        ADD.B   #first-7,R0L
        ADD.B   #7+first,R0L
        ADD.B   #(first/second)*start,R0L
```

SYMBOLS

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

Symbols must begin with a letter, _ (underscore), or ? (question mark). Symbols can include letters and digits, and the symbols _, $, and ?. For user-defined symbols, case is insignificant unless the directive CASEON has been used. For built-in symbols like instructions, registers, operators, and directives case is insignificant.

LABELS

Symbols used for memory locations are referred to as labels.

Location counter
The location counter is called $. For example:

```
BNE     $+3     ; skip over CLC
CLC
BRA     $       ; Never ending loop
```

BIT VARIABLES

The H8 Assembler supports bit addresses for compatibility with the H8 C Compiler, but it is recommended that only the ordinary addressing modes should be used.

The following table shows the bit address ranges for the different H8 processors and memory modes:

Processor	*Memory mode*	*Bit address range*
H8/300H	–	0xFF00 to 0xFFFF
H8S	Small	0x0000 to 0xFFFF
H8S	Large	0x00000000 to 0x00007FFF and 0xFFFF8000 to 0xFFFFFFFF

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 number bases are supported:

Hexadecimal
Hexadecimal numbers can be written in any of the following formats:

Format	*Example*	*Value*
`0x`*`hex-digits`*	`0x12`	18 in decimal.
`H'`*`hex-digits`*	`H'80`	128 in decimal.
`hex-digits``H`	`20H`	32 in decimal*.

* Note that if the first digit is `A–F`, a leading zero must be included; for example, `0AH`.

Octal
Octal numbers can be written as follows:

Format	*Example*	*Value*
`'\`*`octal-digits`*`'`	`'\10'`	8 in decimal.
`Q'`*`octal-digits`*	`Q'10`	8 in decimal.
`octal-digits``Q`	`10Q`	8 in decimal.

Decimal
Decimal numbers can be written as follows:

Format	*Example*	*Value*
`digits`	`123`	123 in decimal.
`D'`*`digits`*	`D'78`	78 in decimal.

Binary
Binary numbers can be written as follows:

Format	*Example*	*Value*
`B'`*`binary-digits`*	`B'10`	2 in decimal.
`binary-digits``B`	`10B`	2 in decimal.

ASCII CHARACTER CONSTANTS

ASCII constants can consist of between zero and four characters enclosed in single 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 (value = 0).
`""`	Empty string (an ASCII null character).
`\'`	`'`
`\\`	`\`

REAL NUMBER CONSTANTS

The H8 Assembler will accept real numbers as constants and convert them into IEEE single-precision (signed 32-bit) real number format.

Floating point numbers can be written in the format:

`[+|-][`*`digits`*`].[`*`digits`*`][{E|e}[+|-]`*`digits`*`]`

Some valid examples are as follows:

Format	*Value*
`10.23`	1.023×10^{1}
`1.23456E-24`	1.23456×10^{-24}
`1.0E3`	1.0×10^{3}

No spaces or tabs are allowed in real constants.

Note that floating-point numbers will not give meaningful results when used in expressions.

PRE-DEFINED SYMBOLS

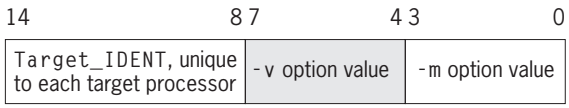
The H8 Assembler defines a set of symbols for use in assembler source files. The symbols provide information about the current assembly, allowing you to test them in pre-processor directives or include them in the assembled code.

Symbol	*Value*
`__DATE__`	Current date in `Mmm dd yyy` format.
`__FILE__`	Current source filename.
`__IAR_SYSTEMS_ASM`	IAR assembler identifier.
`__LINE__`	Current source line number.
`__TID__`	Target identities; see below.
`__TIME__`	Current time in `hh:mm:ss` format.

Target identity symbol

The target identifier contains a number which is unique for each IAR Systems Assembler and C Compiler (ie unique for each target), the value of the `-v` option, and the value corresponding to the `-m` option.

The `__TID__` symbol consists of two bytes:



The `Target_IDENT` for the H8 is `0x25`, and the `-m` option is 0 for `-ms` and 1 for `-ml`.

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 and date of assembly as a string for display by the program:

```
timdat  DC.B    __TIME__,",",__DATE__,0 ; time and date
        ...
        MOV     VVP,timdat      ; load address of string
        JSR     @printstring    ; routine to print string
```

Testing symbols for conditional assembly

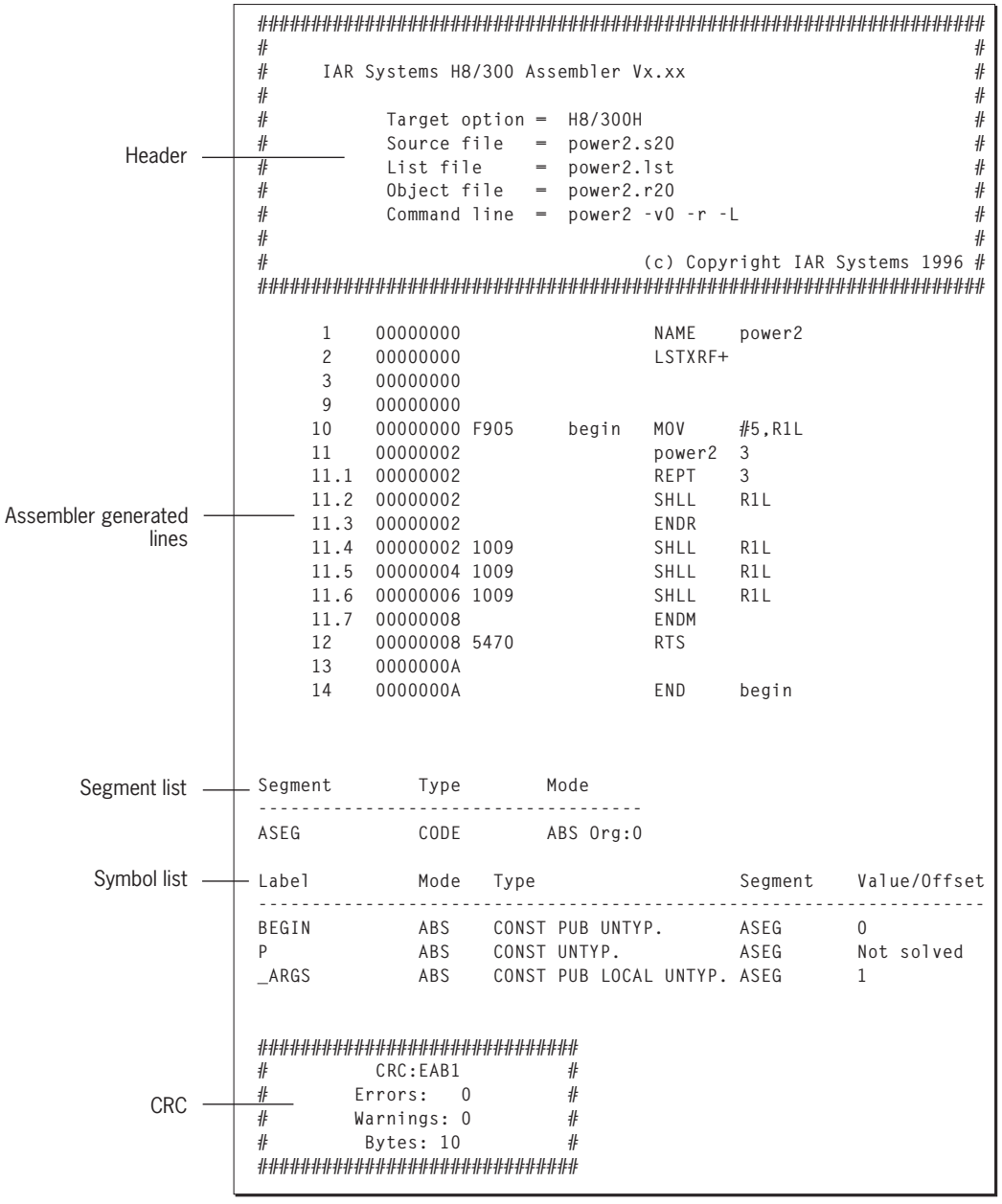
To test a symbol at assembly-time, you use one of the conditional assembly directives.

For example, in a source file written for use on any one of the H8 family members, you might want to assemble appropriate code for a specific processor. You could do this using the __TID__ symbol as follows:

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

LISTING FORMAT

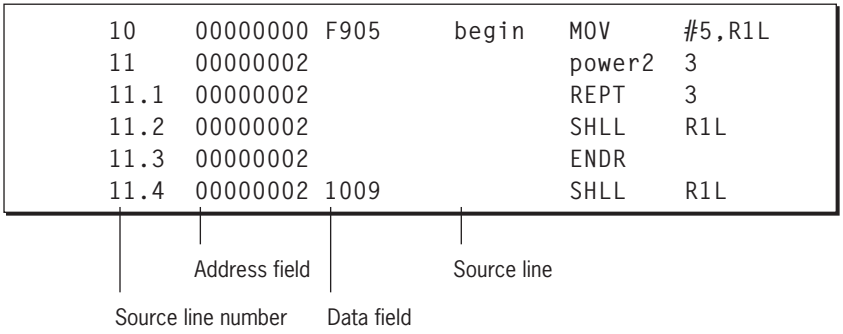
The format of the H8 Assembler listing is as follows:




```
#############################################################################
#                                                                           #
#     IAR Systems H8/300 Assembler Vx.xx                                    #
#                                                                           #
#           Target option =  H8/300H                                        #
#           Source file   =  power2.s20                                     #
#           List file     =  power2.lst                                     #
#           Object file   =  power2.r20                                     #
#           Command line  =  power2 -v0 -r -L                               #
#                                                                           #
#                                        (c) Copyright IAR Systems 1996 #
#############################################################################

      1    00000000                  NAME    power2
      2    00000000                  LSTXRF+
      3    00000000
      9    00000000
     10    00000000 F905     begin   MOV     #5,R1L
     11    00000002                  power2  3
     11.1  00000002                  REPT    3
     11.2  00000002                  SHLL    R1L
     11.3  00000002                  ENDR
     11.4  00000002 1009             SHLL    R1L
     11.5  00000004 1009             SHLL    R1L
     11.6  00000006 1009             SHLL    R1L
     11.7  00000008                  ENDM
     12    00000008 5470             RTS
     13    0000000A
     14    0000000A                  END     begin



Segment         Type        Mode
-------------------------------------
ASEG            CODE        ABS Org:0

Label           Mode   Type                    Segment    Value/Offset
-----------------------------------------------------------------------
BEGIN           ABS    CONST PUB UNTYP.        ASEG       0
P               ABS    CONST UNTYP.            ASEG       Not solved
_ARGS           ABS    CONST PUB LOCAL UNTYP.  ASEG       1


###############################
#          CRC:EAB1           #
#       Errors:   0           #
#       Warnings: 0           #
#        Bytes: 10            #
###############################
```

The header, with assembly parameters, is only output on listings directed to files other than the terminal.

Assembly list information is put into five fields:



Source line number
The line number in the source file.

Lines generated by macros will, if listed, have `.` (full stop) in the source line number field.

Cycle count
The instruction cycle count.

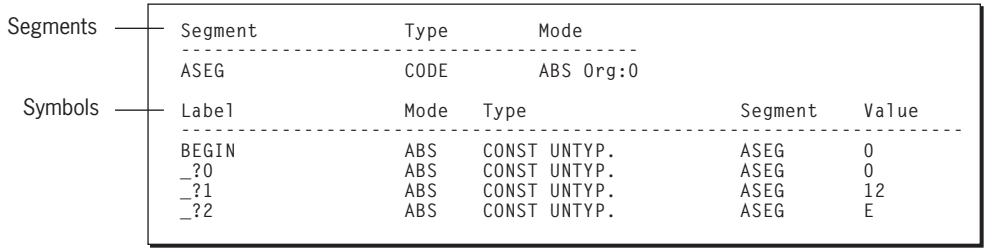
This is only shown if the `LSTCYC+` directive or `-cC` command line option, have been used.

Address and data fields
These are always listed in hexadecimal notation.

Source line
Lists the source file line.

SYMBOL AND CROSS REFERENCE TABLE

If the LSTXRF+ directive has been included, or the -x command line option has been specified, the following symbol and cross reference table is produced:



The following information is provided for each symbol in the table:

Information	*Description*
Label	The label's user-defined name.
Mode	ABS (Absolute), or REL (Relative).
Type	The label's type.
Segment	The name of the segment this label is defined relative to.
Value/Offset	The value (address) of the label within the current module, relative to the beginning of the current segment.

OUTPUT FORMATS

The relocatable and absolute output is in the same format for all assemblers, because object code is always meant to be processed by the IAR Systems XLINK Linker.

The output from XLINK, however, is in absolute formats normally compatible with the chip vendor's debugger programs (monitors), as well as with PROM programmers and stand-alone emulators from independent sources.

ASSEMBLER OPERATOR SUMMARY

This chapter summarizes the assembler operators, classified according to their precedence. A full alphabetical reference list of operators is given in the next chapter, *Assembler operator reference*.

PRECEDENCE OF OPERATORS

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

The following rules determine how expressions are evaluated:

- The highest precedence (lowest number) operators are evaluated first, then the next 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 to group operators and operands and to control the order in which the expressions are evaluated. For example, the following expression evaluates to 1:

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

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

UNARY OPERATORS – 1

-	Unary minus.
+	Unary plus.
NOT	Logical NOT.
LOW	Low byte.
HIGH	Second byte.
BYTE3	Third byte.
LWRD	Low word.
HWRD	High word.
DATE	Current date/time.
SFB	Segment begin.
SFE	Segment end.
SIZEOF	Segment size.
BINNOT	Bitwise NOT.

MULTIPLICATIVE 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.
`XOR`	Logical exclusive OR.
`BINOR`	Bitwise OR.
`BINXOR`	Bitwise exclusive OR.

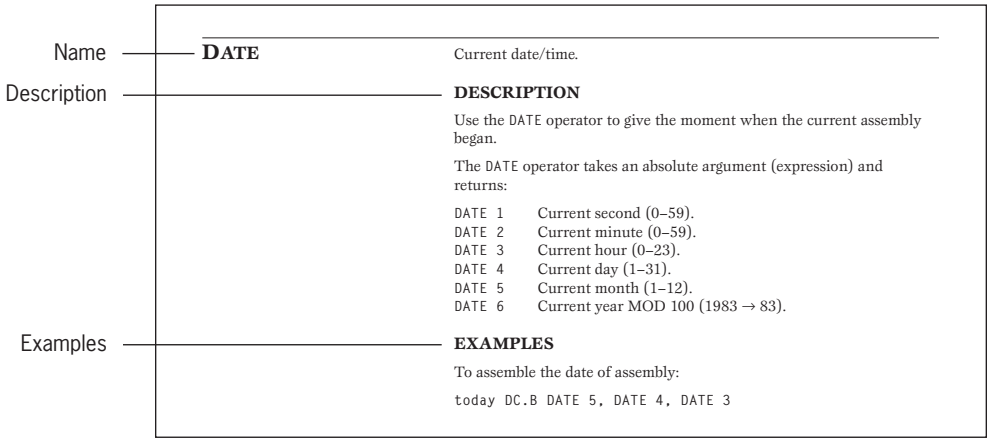
COMPARISON OPERATORS – 7

`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.

ASSEMBLER OPERATOR REFERENCE

This section gives an alphabetical list of the assembler operators with a full description of each one.

The format of each operator description is as follows:



NAME

The operator name, and where appropriate, any synonyms for the operator, and the operator precedence.

The operator name is followed by a description of the operator.

DESCRIPTION

A detailed description covering the operator's most general use.

EXAMPLES

Examples, illustrating typical applications of the operator and clarifying any special cases.

*

Multiplication (3).

DESCRIPTION

* 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.

EXAMPLES

```
2*2 → 4
-2*2 → -4
```

+

Unary plus (1).

DESCRIPTION

Unary plus operator.

EXAMPLES

```
+3 → 3
3*+2 → 6
```

+

Addition (4).

DESCRIPTION

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.

EXAMPLES

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

–

Unary minus (1).

DESCRIPTION

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.

EXAMPLES

```
-2--2 → 0
```

–

Subtraction (4).

DESCRIPTION

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.

EXAMPLES

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

/

Division (3).

DESCRIPTION

/ 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.

EXAMPLES

```
8/2 → 4
-12/3 → -4
```

AND

Logical AND (5).

DESCRIPTION

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

EXAMPLES

```
1010B AND 0011B → 1
1010B AND 0101B → 1
1010B AND 0000B → 0
```

BINAND

Bitwise AND (5).

DESCRIPTION

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

EXAMPLES

```
1010B BINAND 0011B → 0010B
1010B BINAND 0101B → 0000B
1010B BINAND 0000B → 0000B
```

BINNOT

Bitwise NOT (1).

DESCRIPTION

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

EXAMPLES

```
BINNOT 1010B → 11111111111111111111111111110101B
```

BINOR

Bitwise OR (6).

DESCRIPTION

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

EXAMPLES

```
1010B BINOR 0101B → 1111B
1010B BINOR 0000B → 1010B
```

BINXOR

Bitwise exclusive OR (6).

DESCRIPTION

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

EXAMPLES

```
1010B BINXOR 0101B → 1111B
1010B BINXOR 0011B → 1001B
```

BYTE3

Third byte (1).

DESCRIPTION

`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.

EXAMPLES

```
BYTE3 0x12345678 → 0x34
```

DATE

Current date/time.

DESCRIPTION

Use the DATE operator to give the moment 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 (1983 → 83).

EXAMPLES

To assemble the date of assembly:

```
today DC.B DATE 5, DATE 4, DATE 3
```

EQ (=)

Equal (7).

DESCRIPTION

EQ 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.

EXAMPLES

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

GE (>=)

Greater than or equal (7).

DESCRIPTION

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

EXAMPLES

```
1 GE 2 → 0
2 GE 1 → 1
1 GE 1 → 0
```

GT (>)

Greater than (7).

DESCRIPTION

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

EXAMPLES

```
-1 GT 1 → 0
2 GT 1 → 1
1 GT 1 → 0
```

HIGH

Second byte (1).

DESCRIPTION

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.

EXAMPLES

```
HIGH 1234ABCDh → ABh
```

HWRD

High word (1).

DESCRIPTION

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.

EXAMPLES

```
HWRD 0x12345678 → 0x1234
```

LE (<=)

Less than or equal (7).

DESCRIPTION

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

EXAMPLES

```
1 LE 2 → 1
2 LE 1 → 0
1 LE 1 → 1
```

LOW

Low byte (1).

DESCRIPTION

LOW takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the unsigned, 8-bit integer value of the lower order byte of the operand.

EXAMPLES

```
LOW 1234ABCDh → CDh
```

LT (<)

Less than (7).

DESCRIPTION

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

EXAMPLES

```
-1 LT 2 → 1
2 LT 1 → 0
2 LT 2 → 0
```

LWRD

Low word (1).

DESCRIPTION

`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.

EXAMPLES

```
LWRD 0x12345678 → 0x5678
```

MOD (%)

Modulo (3).

DESCRIPTION

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

EXAMPLES

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

NE (<>)

Not equal (7).

DESCRIPTION

NE 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.

EXAMPLES

```
1 NE 2 → 1
2 NE 2 → 0
'A' NE 'B' → 1
```

NOT

Logical NOT (1).

DESCRIPTION

Use NOT to negate a logical argument.

EXAMPLES

```
NOT 0101B → 0
NOT 0000B → 1
```

OR (|)

Logical OR (6).

DESCRIPTION

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

EXAMPLES

```
1010B OR 0000B → 1
0000B OR 0000B → 0
```

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.

EXAMPLES

```
        NAME  demo
        RSEG  CODE
start   SET   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.

EXAMPLES

```
      NAME  demo
      RSEG  CODE
end   SET   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.

SHL (<<)

Logical shift left (3).

DESCRIPTION

Use SHL to shift the left operand to the left. The number of bits to shift is specified by the right operand, interpreted as an integer value between 0 and 32.

EXAMPLES

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

SHR (>>)

Logical shift right (3).

DESCRIPTION

Use SHR to shift the left operand to the right. The number of bits to shift is specified by the right operand, interpreted as an integer value between 0 and 32.

EXAMPLES

```
01110000B SHR 3 → 00001110B
1111111111111111B SHR 20 → 0
14 SHR 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; ie it calculates the size in bytes of a segment. This is done when modules are linked together.

EXAMPLES

```
      NAME   demo
      RSEG   CODE
size  SET    SIZEOF CODE
```

sets size to the size of segment CODE.

UGT

Unsigned greater than (7).

DESCRIPTION

UGT evaluates to 1 (true) if the left operand has a larger absolute value than the right operand.

EXAMPLES

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

ULT

Unsigned less than (7).

DESCRIPTION

`ULT` evaluates to 1 (true) if the left operand has a smaller absolute value than the right operand.

EXAMPLES

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

XOR

Logical exclusive OR (6).

DESCRIPTION

Use `XOR` to perform logical XOR on its two operands.

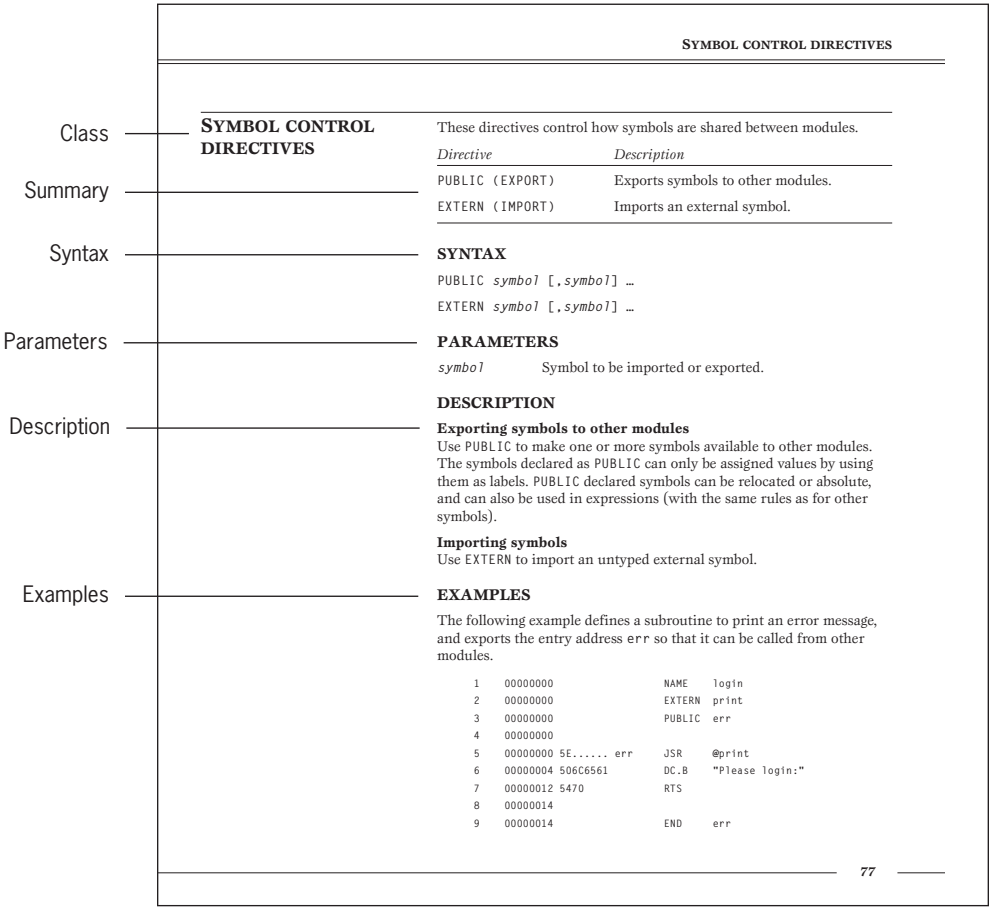
EXAMPLES

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

ASSEMBLER DIRECTIVES REFERENCE

This chapter gives a list of the H8 directives, classified according to their function, with a full description of their operation, and the options available for each one.

The format of each section is as follows:



CLASS

The class of directives.

SUMMARY

The class is followed by a summary of the class, and a description of each directive in the class.

SYNTAX

A full syntax definition of each directive.

PARAMETERS

Details of each parameter in the syntax definitions.

DESCRIPTION

A detailed description covering each directive's most general use. This includes information about what the directives are useful for, and a discussion of any special conditions and common pitfalls.

EXAMPLES

Examples, illustrating typical applications of the directives and clarifying any special cases.

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:

```
LOCAL symbol [,symbol] …
```

indicates that LOCAL 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 directive must be preceded by a label, this is indicated in the syntax, as in:

```
label SET expr
```

All other directives can be preceded by an optional label, which will assume the value and type of the current location counter (PLC), and 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 ; (semi-colon).

PARAMETERS

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

Parameter	*What it consists of*
symbol	An assembler symbol.
label	A symbolic label.
expr	An expression; see *Expressions and operators*, page 43.

MODULE CONTROL DIRECTIVES

Module control directives are used to mark the beginning and end of source program modules, and to assign names and types to them.

Directive	*Description*
`NAME (PROGRAM)`	Begins a program module.
`MODULE (LIBRARY)`	Begins a library module.
`ENDMOD`	Terminates the assembly of the current module.
`END`	Terminates the assembly of the last module in a file.

SYNTAX

```
NAME symbol [(expr)]

MODULE symbol [(expr)]

ENDMOD [label]

END [label]
```

PARAMETERS

symbol	Name assigned to module, used by XLIB when referencing the module.
expr	Optional expression (0–255) used by the IAR C Compiler.
label	An expression or label which can be resolved at assembly time. It is output in the object code as a program entry address.

DESCRIPTION

Beginning a program module

Use `NAME` to begin a program module, and assign a name for future reference by XLINK and XLIB.

Program modules are unconditionally linked by XLINK, even if they are not referenced by other modules.

Beginning a library module
Use MODULE to create libraries containing lots of small modules (like run time systems for high level languages), where each module also often represent 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 a public symbol in the module is referenced by other modules.

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.

Program entries must be either relocatable or absolute (no externals allowed), and will show up in XLINK load maps, as well as in some of the hexadecimal absolute output formats.

The following rules apply to multi-module assemblies:

- 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.
- List control directives remain in effect throughout the assembly.

Note that END must always be used in the *last* module, and that there must not be any source lines (except for comments and list 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.

EXAMPLES

The following example defines three modules:

```
MODULE
.
. Module #1
.
ENDMOD
MODULE
.
. Module #2
```

```
.
ENDMOD
MODULE
.
. Last module
.
END
```

SYMBOL CONTROL DIRECTIVES

These directives control how symbols are shared between modules.

Directive	*Description*
PUBLIC (EXPORT)	Exports symbols to other modules.
EXTERN (IMPORT)	Imports an external symbol.

SYNTAX

PUBLIC *symbol* [,*symbol*] …

EXTERN *symbol* [,*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. PUBLIC declared symbols 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 and 16-bit processors. With the LOW, HIGH, LWRD, and HWRD, operators any part of such a constant can be loaded in an 8- 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.

```
1    00000000                  NAME    login
2    00000000                  EXTERN  print
3    00000000                  PUBLIC  err
4    00000000
5    00000000 5E...... err     JSR     @print
6    00000004 506C6561         DC.B    "Please login:"
7    00000012 5470             RTS
8    00000014
9    00000014                  END     err
```

SEGMENT CONTROL DIRECTIVES

The segment directives control how code and data are generated.

Directive	*Description*
ASEG	Begins an absolute segment.
RSEG	Begins a relocatable segment.
STACK	Begins a stack segment.
COMMON	Begins a common segment.
ORG	Sets the location counter.
ALIGN	Aligns the program counter by inserting zero-filled bytes.
EVEN	Aligns the program counter to an even address by inserting a zero-filled byte.

SYNTAX

```
ASEG [start [(align)]]

RSEG segment [:type] [(align)]

STACK segment [:type] [(align)]

COMMON segment [:type] [(align)]

ORG expr

ALIGN align [,code]

EVEN
```

PARAMETERS

start	A start address which has the same effect as using an ORG directive at the beginning of the absolute segment.
segment	The name of the segment.
type	The memory type; one of: UNTYPED (the default), CODE, or DATA. In addition, the following types are provided for compatibility with the IAR C Compilers: XDATA, IDATA, BIT, REGISTER, and CONST.
expr	Address to set location counter to.
align	Power of two to which the address should be aligned, in the range 0 to 30.
code	Code to be inserted when aligning; defaults to 0.

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 *start* 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 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 256 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 (vs. the RSEG directive which causes low-to-high allocation).

Note that 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 where you desire to have a number of different routines 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 *Segment control*, page 179.

Specifying the *align* parameter in any of the above directives is equivalent to including an ALIGN directive with the same value.

Setting the location counter
Use ORG to set the 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, that is, it is not valid to use ORG 10 during RSEG, since the expression is absolute; instead use ORG $+10. The expression must not contain any forward or external references.

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

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

Use EVEN to align the program counter to an even address. A 0 is inserted if necessary. It is equivalent to ALIGN 1.

EXAMPLES

Beginning an absolute segment
The following example makes the subroutine subr start in a next new page after address 123:

```
10    0000007B          subr    ASEG    123 (8)
11    00000100 F90A             MOV     #10,R1L
12    00000102 5090             MULXU   R1L,R0
13    00000104 5470             RTS
14    00000106
15    00000106                  END     main
```

After assembling this code sseg will have the value 7B and subr will have the value 100.

Beginning a relocatable segment
The following directive aligns the start address of segment MYSEG (upwards) to the nearest 8 byte (2**3) page boundary:

```
RSEG MYSEG :CODE(3)
```

Note that only the *first* segment directive for a particular segment can contain an alignment operand.

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

```
        STACK   rpnstack
parms   DS.B    100
opers   DS.B    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   DS.B      4
        ENDMOD

        NAME    common2
        COMMON  data
up      DS.B      1
        ORG     $+2
down    DS.B      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`.

Setting the location counter
The following example uses `ORG` to leave a gap of 256 bytes:

```
        NAME    org
        ORG     $+256
begin   MOV     #12,R3L
        MULXU   R3L,R2
        RTS
        END     begin
```

Aligning a segment
The following example uses an `ALIGN` directive to ensure that a subroutine starts on a page boundary:

```
 1    00000000                NAME    align
 2    00000000
 3    00000000 F807     main  MOV     #7,R0L
 4    00000002 5E000100       JSR     @subr
 5    00000006 0D10           MOV     R1,R0
 6    00000008 5470           RTS
 7    0000000A
 8    0000000A          ; Make subr start in a new page
 9    0000000A 00000000       ALIGN   8
10    00000100 F90A     subr  MOV     #10,R1L
```

```
11    00000102 5090             MULXU   R1L,R0
12    00000104 5470             RTS
13    00000106
14    00000106                  END     main
```

The following example uses EVEN to start a table on an even address:

```
 1    00000000                  NAME    even
 2    00000000
 3    00000000 01               DC.B    1
 4    00000001 00               EVEN
 5    00000002 0001000A table   DC.W    1,10,100
 6    00000008                  END
```

VALUE ASSIGNMENT DIRECTIVES

These directives are used to assign values to symbols.

Directive	*Description*
SET (VAR, ASSIGN)	Assigns a temporary value.
EQU (=)	Assigns a permanent value local to a module.
DEFINE	Defines a file-wide value.
SFR	Creates byte-access SFR labels.
SFRP	Creates word-access SFR labels.
SFRTYPE	Specifies SFR attributes.
LIMIT	Checks that values lie within a specified range.

SYNTAX

label SET *expr*

label EQU *expr*

label = *expr*

label DEFINE *expr*

[const] SFR *register* = *value*

[const] SFRP *register* = *value*

```
[const] SFRTYPE register attribute [,attribute] = value
LIMIT label, min, max, "message"
```

PARAMETERS

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

DESCRIPTION

Defining a temporary value
Use SET to define a symbol which may be redefined, such as for use with macro variables. Symbols defined with SET 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.

To import symbols from other modules use EXTERN.

Defining a permanent global value
Use DEFINE to define symbols that should be known to all modules in the source file.

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

Symbols defined with DEFINE 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, and WORD turned on. Use SFRTYPE to create special function register labels with specified attributes.

Note that all SFR labels must be specified with a full 32-bit address. The high word is ignored on processors with a 64 Kbyte address range.

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.

The include files ioh8xxx.h provided on the distribution disk use the SFRTYPE directive to declare the attributes for the standard SFRs. This creates untyped labels, but with additional attributes to allow the H8 Assembler to check their usage and report errors if they are used incorrectly.

Checking a symbol
Use LIMIT to check the value of a symbol.

The check will occur as soon as the value 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, ie they must be resolved when encountered.

EXAMPLES

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

```
        NAME    table

main    ; Generate table of powers of 3
const   SET     1
        REPT    8
```

```
        DC.W    const
const   SET     const * 3
        ENDR
        END     main
```

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`:

```
        NAME    add1
locn    DEFINE  100H
value   EQU     77
        MOV     @locn,R1L
        ADD     #value,R1L
        ENDMOD

        NAME    add2
value   EQU     88
        MOV     @locn,R2L
        ADD     #value,R2L
        END
```

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

Using SFRs
The following example defines two `SFR`s:

```
        SFR SYSCR = 0xFFFFFFF2
        SFR MDCR  = 0xFFFFFFF7
```

The following shows how attributes can be combined in any manner to express how the SFR can be used:

```
        SFRTYPE AD_AMR byte,read,write = 0xFFFFFFC4
        SFRTYPE AD_APR byte,read       = 0xFFFFFFC5
```

However the following instruction is illegal since `BYTE` access is not used, and the assembler will give a warning:

```
        MOV     R0,P0
```

Checking a symbol
The following example shows a label `lab` being range-checked:

```
1         0000000             LIMIT     lab,10,12,"My fault"
2         000000A   lab       SET       10
3         000000B   lab       SET       lab+1 ;lab=11,ok
4         000000C   lab       SET       lab+1 ;lab=12,ok
5                   lab       SET       lab+1 ;lab=13,error My fault
6         0000000             END
```

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.
`ENDIF`	Ends an `IF` block.

SYNTAX

`IF` *`condition`*

`ELSE`

`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 of *`string1`* and *`string2`* have different length or contents.

DESCRIPTION

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 (ie it will not be assembled or syntax checked) until an ELSE or ENDIF directive is found.

Conditional assembler directives may be used anywhere in an assembly, but have their greatest use in conjunction with macro processing.

All assembler directives (except for END), and file inclusion, may be disabled by the conditional directives. Each IF*xx* directive must be terminated by an 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.

EXAMPLES

The following macro assembles instructions to multiply ROL by a constant, but omits them if the argument is 1:

```
        NAME    mult

mult    MACRO   k
        IF      k <> 1
        MOV     #k,R1L
        MULXU   R1L,R0
        ENDIF
        ENDM
```

It could be tested with the following program:

```
main    MOV     #23,R0L
        mult    7
        END     main
```

MACRO PROCESSING DIRECTIVES

These directives allow user macros to be defined.

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

SYNTAX

name MACRO [*argument*] …

ENDM

EXITM

LOCAL *symbol* [,*symbol*] …

REPT *expr*

REPTC *formal*,*actual*

REPTI *formal*,*actual* [,*actual*] …

ENDR

PARAMETERS

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

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 just 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.

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

Defining a macro
You define a macro with the statement:

macroname MACRO [*argument*] [*argument*] …

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

For example, you could define a macro `getbyte` as follows:

```
getbyte  MACRO    port
         JSR      @waitdata
         MOV      port,R0L
         ENDM
```

This uses a parameter `port` to specify the port address to read from. You would call the macro with a statement such as:

```
         getbyte  #0x8000
```

This will be expanded by the assembler to:

```
         JSR      @waitdata
         MOV      #0x8000,R0L
```

If you omit a list of one or more arguments, the arguments you supply when calling the macro are called `\1` to `\9` and `\A` to `\Z`.

The previous example could therefore be written as follows:

```
error    MACRO
         JSR      @waitdata
         MOV      \1,R0L
         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.

It is illegal to *redefine* a macro.

Creating local symbols
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.

Finding the number of arguments
The macro-local symbol _args provides the number of arguments the macro was called with.

For example, the following macro checks that there are at most 10 arguments:

```
chcount MACRO   parm
        IF      _args>10
        EXITM
        ENDIF
        DC.B    _args
        ENDM
```

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:

```
mac1    MACRO   regs
        ADD     regs    ; add content of reg1 to reg2
        ENDM
```

It could be called using:

```
        mac1    <R1L,R2L>
        END
```

You can redefine the macro quote characters with the -M command line option; see *Macro quote chars (-M)*, page 37.

How macros are processed

There are three distinct phases in the macro process:

- Scanning and saving of macro definitions is performed by the assembler. 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) which switches (if not already in a macro) the assembler input stream from a source file to the output from the macro expander (which 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.

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.

Extending the instruction set

Because of the way in which microprocessor instruction sets evolve, they are not always as symmetrical as one would like. By writing macros you can add definitions for instructions that you would like to have

included in the instruction set, and use them just like the built-in instructions.

For example, the SUB instruction does not support immediate data for byte operands. If you frequently need this operation you could define a subi macro to do this as follows:

```
subi    MACRO  immed,reg
        ORC    #H'05,CCR
        SUBX   #(immed-1),reg
        ENDM
```

This could then be used in a program as follows:

```
main    subi    27,R0L
        RTS

        END
```

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.

For example, the following subroutine outputs a 256-byte buffer to a port:

```
        EXTERN  port

        RSEG    RAM
buffer  DS.B    256

        RSEG    PROM
; Plays 256 bytes from buffer to port
play    MOV     #0,R1
loop    MOV     @(buffer,R1),R2L
        MOV     R2L,@port
        INC     R1L
        BNE     loop
        RTS

        END
```

The main program calls this routine as follows:

```
doplay  JSR     @play
```

For efficiency we can recode this as the following macro:

```
; Plays 256 bytes from buffer to port
play    MACRO
        LOCAL   loop
        MOV     #0,R1
loop    MOV     @(buffer,R1),R2L
        MOV     R2L,@port
        INC     R1L
        BNE     loop
        ENDM

        END
```

Note the use of `LOCAL` 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.

To use in-line code the main program is then simply altered to:

```
doplay play
```

Exiting from a macro

The following example defines a macro to rotate register `R1L` a specified number of times, `r`. It uses `EXITM` to exit from the macro if `r` is `8`, since no rotates are needed:

```
        NAME    rotate

rotate  MACRO   r
        IF      r = 8
        EXITM
        ENDIF
        REPT    r
        ROTR    R1L
        ENDR
        ENDM
```

Using REPT

The following example uses `REPT` to assemble a table of powers of 3:

```
        NAME    table

main    ; Generate table of powers of 3
calc    SET     1
        REPT    8
```

```
        DC.W    calc
calc    SET     calc * 3
        ENDR
        END     main
```

It generates the following code:

```
1    00000000                NAME    table
2    00000000
3    00000000          main  ; Generate table of powers of 3
4    00000001          calc  SET     1
5    00000000                REPT    8
6    00000000                DC.W    calc
7    00000000          calc  SET     calc * 3
8    00000000                ENDR
8.1  00000000 0001           DC.W    calc
8.2  00000003          calc  SET     calc * 3
8.3  00000002 0003           DC.W    calc
8.4  00000009          calc  SET     calc * 3
8.5  00000004 0009           DC.W    calc
8.6  0000001B          calc  SET     calc * 3
8.7  00000006 001B           DC.W    calc
8.8  00000051          calc  SET     calc * 3
8.9  00000008 0051           DC.W    calc
8.10 000000F3          calc  SET     calc * 3
8.11 0000000A 00F3           DC.W    calc
8.12 000002D9          calc  SET     calc * 3
8.13 0000000C 02D9           DC.W    calc
8.14 0000088B          calc  SET     calc * 3
8.15 0000000E 088B           DC.W    calc
8.16 000019A1          calc  SET     calc * 3
9    00000010                END     main
```

Using REPTC and REPT1

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

```
prompt  REPTC   char,"Login:"
        MOV     #char,R0L
        JSR     @putc
        ENDR
```

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

```
main    MOV     #0,ROL
        REPTI   zero,flag,temp,"(base,R1)"
        MOV     ROL,@zero
                ENDR
```

STRUCTURED ASSEMBLY DIRECTIVES

The structured assembly directives allow loops and control structures to be implemented at assembly level.

Directive	*Description*
IFS	Specifies instructions to be executed if a condition is true.
ELSES	Introduces instructions to be executed if a condition is false.
ELSEIFS	Specifies a new condition in an IFS block.
ENDIFS	Ends an IFS block.
WHILE	Repeats subsequent instructions while a condition is true.
ENDW	Ends a WHILE loop.
REPEAT	Repeats subsequent instructions until a condition is true.
UNTIL	Ends a REPEAT loop.
FOR	Repeats subsequent instructions a specified number of times.
ENDF	Ends a FOR loop.
SWITCH	Multiple case switch.
CASE	Case in SWITCH block.
DEFAULT	Default case in SWITCH block.
ENDS	Ends a SWITCH block.
BREAK	Exits prematurely from a loop or switch construct.
CONTINUE	Continues execution of a loop or switch construct.

SYNTAX

```
IFS[.size] test [AND test | OR test] THEN
ELSE
ELSEIFS[.size] test [AND test | OR test] THEN
ENDIF

WHILE[.size] test [AND test | OR test] THEN
ENDW

REPEAT
UNTIL[.size] test [AND test | OR test]

FOR[.size] reg=start {TO | DOWNTO} end [BY step] DO
ENDF

SWITCH[.size] op1
CASE op2
DEFAULT
ENDS

BREAK
CONTINUE
```

PARAMETERS

size — One of `.B`, `.W`, or `.L` to specify the size for the `CMP` and `MOV` instructions. If omitted, `.B` is assumed.

test — A comparison of the form:

op1 condition op2

where *op1* and *op2* can have the following forms:

Size	*op1*	*op2*
.B	Rs	Rd #xx:8
.W	Rs	Rd #xx:16
.L	ERs	ERd #xx:32

condition should be chosen from the following table:

Comparison	*Unsigned*	*Signed*
op1<>*op2*	<NE> Not equal	<NE> Not equal
op1<*op2*	<CS> Carry set <LO> Lower	<LT> Less than
op1<=*op2*	<LS> Lower or same	<LE> Less than or equal
op1=*op2*	<EQ> Equal	<EQ> Equal
op1>=*op2*	<CC> Carry clear <HS> Higher or same	<GE> Greater than or equal
op1>*op2*	<HI> Higher	<GT> Greater than

or the following additional conditions:

Comparison	*Condition*
N=0	<PL> Plus
N=1	<MI> Minus
V=0	<VC> Overflow clear
V=1	<VS> Overflow set

reg, *start*, *end*, *step* — Operands with one of the forms shown in the following tables:

Size	*reg*	*start*	*end*	*step*
.B	Rd	#xx:8 Rs @ERs @(d:16,ERs) @ERs+ @aa:8 @aa:16 @aa:24/32 †	#xx:8 Rd	#xx:8 Rs

Size	*reg*	*start*	*end*	*step*

`.W`	`Rd`	`#xx:16`
		`Rs`
		`@ERs`
		`@(d:16,ERs)`
		`@(d:24/32,ERs)` †
		`@ERs+`
		`@aa:16`
		`@aa:24/32` †
			`#xx:16`
			`Rd`
				`#xx:16`
				`Rd`

`.L`	`ERd`	`#xx:32`
		`ERs`
		`@ERs`
		`@(d:16,ERs)`
		`@(d:24/32,ERs)` †
		`@ERs+`
		`@aa:16`
		`@aa:24/32` †
			`#xx:32`
			`Rd`
				`#xx:32`
				`Rs`

† 24 for the H8/300H, 32 for the H8S.

If *`step`* is omitted it defaults to 1, or -1 if `DOWNTO` is specified.

DESCRIPTION

The H8 Assembler includes a versatile range of directives for structured assembly, to make it easier to implement loops and control structures at assembly level.

The advantage of using the structured assembly directives is that the resulting programs are clearer, and their logic is easier to understand.

The directives are designed to generate simple, predictable code so that the resulting program is as efficient as if it were programmed by hand.

Conditional constructs

Use `IFS` … `ENDIFS` to generate assembler source-code for comparison and jump instructions. The generated code is assembled like ordinary code, and is similar to macros. This should not be confused with conditional assembly.

`IFS` blocks can be nested to any level.

The IFS directives generates the statements:

```
CMP.size op2,op1
Binverted label
```

where *inverted* is the inverted *condition*.

Use ELSES after an IFS directive to introduce instructions to be executed if the IFS condition is false.

Use ELSEIFS to introduce a new condition after an IFS directive.

Loop directives
Use WHILE ... ENDW to create a loop which is executed as long as the expression is TRUE. If the expression is false at the beginning of the loop the body will not be executed.

Use the REPEAT ... UNTIL construct to create a loop with a body that is executed at least once, and as long as the expression is FALSE.

You can use BREAK to exit prematurely from a WHILE ... ENDW or REPEAT ... UNTIL loop, or CONTINUE to continue with the next iteration of the loop.

The directives generate the same statements as the IFS directive.

Iteration construct
Use FOR to assemble instructions to repeat a block of instructions for a specified sequence of values.

FOR.B generates the following instructions:

```
        MOV.B   start,reg
        BRA     compare
inc     ADD     step
compare CMP.B   end,reg
        BLT     out
        BRA     inc
out
```

If *step* is omitted an INC instruction is used instead of ADD.

If DOWNTO is specified instead of TO the INC changes to DEC, and ADD changes to SUBX if *step* is not an immediate value.

FOR.W generates the following instructions:

```
        MOV.W   start,reg
        BRA     compare
```

```
inc     ADD.W   step
compare CMP.W   end,reg
        BLT     out
        BRA     inc
out
```

If *step* is omitted the ADD changes to ADDS#1.

If DOWNTO is specified instead of TO the ADD changes to SUBX if *step* is not an immediate value.

BREAK can be used to exit prematurely from a FOR loop, and continue execution at the instruction following the ENDF.

CONTINUE can be used to continue with the next iteration of the loop.

Switch construct
Use the SWITCH ... ENDS block to execute one of a number of sets of statements, depending on the value of a test.

CASE defines each of the tests, and DEFAULT introduces a CASE which is always true.

Note that CASE falls through by default as in C. CASEIN is similar to CASE except that it checks a range, where *op1* must be less than or equal to *op2*.

BREAK can be used to exit from a SWITCH ... ENDS block.

The SWITCH construct generates the following code:

The following statements are generated:

```
        CMP.size op2,op1
        BNE label
```

EXAMPLES

Using conditional constructs
The following program example tests a memory location and sets R1L to 'N', 'P', or 'Z', depending on whether it is less than zero, zero, or greater than zero:

```
        NAME    test
        EXTERN  value

main    MOV     @value,R0L
        IFS     R0L <MI> #0 THEN
        MOV     #'N',R1L
```

```
        ELSES
        IFS     R0L <EQ> #0 THEN
        MOV     #'Z',R1L
        ELSES
        MOV     #'P',R1L
        ENDIFS
        ENDIFS

        END     main
```

This generates the following code:

```
   1    00000000                NAME    test
   2    00000000                EXTERN  value
   3    00000000
   4    00000000 6A08.... main  MOV     @value,R0L
   5    00000004                IFS     R0L <MI> #0 THEN
 5.1    00000004 A800           CMP #0,R0L
 5.2    00000006 4A04           BPL _?0
   6    00000008 F94E           MOV     #'N',R1L
   7    0000000A                ELSES
 7.1    0000000A 400A           BRA _?1
 7.2    0000000C         _?0
   8    0000000C                IFS     R0L <EQ> #0 THEN
 8.1    0000000C A800           CMP #0,R0L
 8.2    0000000E 4604           BNE _?2
   9    00000010 F95A           MOV     #'Z',R1L
  10    00000012                ELSES
10.1    00000012 4002           BRA _?3
10.2    00000014         _?2
  11    00000014 F950           MOV     #'P',R1L
  12    00000016                ENDIFS
12.1    00000016         _?3
  13    00000016                ENDIFS
13.1    00000016         _?1
  14    00000016
  15    00000016                END     main
```

It can be rewritten more concisely using `ELSEIFS` as follows:

```
        NAME    test
        EXTERN  value

main    MOV     @value,R0L
        IFS     R0L <MI> #0 THEN
```

```
        MOV     #'N',R1L
        ELSEIFS R0L <EQ> #0 THEN
        MOV     #'Z',R1L
        ELSES
        MOV     #'P',R1L
        ENDIFS
        END     main
```

This generates the following code:

```
 1    00000000                NAME    test
 2    00000000                EXTERN  value
 3    00000000
 4    00000000 6A08.... main  MOV     @value,R0L
 5    00000004                IFS     R0L <MI> #0 THEN
5.1   00000004 A800           CMP #0,R0L
5.2   00000006 4A04           BPL _?0
 6    00000008 F94E           MOV     #'N',R1L
 7    0000000A                ELSES
7.1   0000000A 400A           BRA _?1
7.2   0000000C          _?0
 8    0000000C                IFS     R0L <EQ> #0 THEN
8.1   0000000C A800           CMP #0,R0L
8.2   0000000E 4604           BNE _?2
 9    00000010 F95A           MOV     #'Z',R1L
10    00000012                ELSES
10.1  00000012 4002           BRA _?3
10.2  00000014          _?2
11    00000014 F950           MOV     #'P',R1L
12    00000016                ENDIFS
12.1  00000016          _?3
13    00000016                ENDIFS
13.1  00000016          _?1
14    00000016
15    00000016                END     main
```

Using loop constructs

The following example uses a `REPEAT … UNTIL` loop to reverse the order of bits in register `R0L` and put the result in `R1L`:

```
        NAME    until

reverse MOV     #0,R1L
        REPEAT
```

```
        SHLR    R0L
        ROTXL   R1L
        UNTIL   R0L <EQ> #0
        RTS

        END     reverse
```

This generates the following code:

```
     1    00000000                NAME    until
     2    00000000
     3    00000000 F900    reverse MOV     #0,R1L
     4    00000002                REPEAT
   4.1    00000002         _?0
     5    00000002 1108           SHLR    R0L
     6    00000004 1209           ROTXL   R1L
     7    00000006                UNTIL   R0L <EQ> #0
   7.1    00000006 A800           CMP #0,R0L
   7.2    00000008 46F8           BNE _?0
   7.3    0000000A         _?1
     8    0000000A 5470           RTS
     9    0000000C
    10    0000000C                END     reverse
```

Using FOR … NEXT

The following example uses a FOR block to output a 256-byte buffer to a port:

```
        NAME    for
        EXTERN  ioport,outbuf

        RSEG    prom
        MOV     #0,R0H
play    FOR     R0L = #0 TO #255 DO
        MOV     @(outbuf,R0),R1L
        MOV     R1L,@ioport
        ENDF
        RTS

        END
```

It generates the following code:

```
     1    00000000                NAME    for
     2    00000000                EXTERN  ioport,outbuf
     3    00000000
```

```
 4    00000000                  RSEG    prom
 5    00000000 F000             MOV     #0,R0H
 6    00000002          play    FOR     R0L = #0 TO #255 DO
6.1   00000002 F800             MOV #0,R0L
6.2   00000004 4002             BRA _?0
6.3   00000006 0A08     _?1     INC R0L
6.4   00000008 A8FF     _?0     CMP #255,R0L
6.5   0000000A 4E0A             BGT _?2
 7    0000000C 6E09....         MOV     @(outbuf,R0),R1L
 8    00000010 6A89....         MOV     R1L,@ioport
 9    00000014                  ENDF
9.1   00000014 40F0             BRA _?1
9.2   00000016          _?2
10    00000016 5470             RTS
11    00000018
12    00000018                  END
```

Using switch constructs

The following example uses a `SWITCH … ENDS` block to print `Zero`, `Positive`, or `Negative` depending on the value of the `R1L` register. It uses an external `print` routine to print an immediate string:

```
        NAME    switch
        EXTERN  print

test    AND     #H'81,R1L
        SWITCH  R1L

        CASE    #0
        JSR     @print
        DC.B    "Zero"
        BREAK

        CASE    #H'80
        CASE    #H'81
        JSR     @print
        DC.B    "Negative"
        BREAK

        DEFAULT
        JSR     @print
        DC.B    "Positive"

        ENDS

        END     test
```

This generates the following code:

```
 1    00000000                  NAME    switch
 2    00000000                  EXTERN  print
 3    00000000
 4    00000000 E981     test    AND     #H'81,R1L
 5    00000002                  SWITCH  R1L
 6    00000002
 7    00000002                  CASE    #0
7.1   00000002 A900     _?0     CMP #0,R1L
7.2   00000004 460B             BNE _?2
 8    00000006 5E......         JSR     @print
 9    0000000A 5A65726F         DC.B    "Zero"
10    0000000F                  BREAK
10.1  0000000F 4024             BRA _?1
11    00000011
12    00000011                  CASE    #H'80
12.1  00000011 A980     _?2     CMP #H'80,R1L
12.2  00000013 4600             BNE _?3
13    00000015                  CASE    #H'81
13.1  00000015 A981     _?3     CMP #H'81,R1L
13.2  00000017 460F             BNE _?4
14    00000019 5E......         JSR     @print
15    0000001D 4E656761         DC.B    "Negative"
16    00000026                  BREAK
16.1  00000026 400D             BRA _?1
17    00000028
18    00000028                  DEFAULT
18.1  00000028          _?4
19    00000028 5E......         JSR     @print
20    0000002C 506F7369         DC.B    "Positive"
21    00000035
22    00000035                  ENDS
22.1  00000035          _?1
23    00000035
24    00000035                  END     test
```

The following example demonstrates the use of `BREAK` and `CONTINUE` in a `SWITCH` … `ENDS` block:

```
NAME   break
EXTERN  print
ASEG    0
```

```
        DC.W    start   ;reset vector

        ORG     100
start   MOV     #1,R1L
        JSR     @print_msg
        BRA     $       ;wait forever

print_msg
        SWITCH  R1L

        CASE    #0      ;print both messages
        MOV     #m1,R0
        JSR     @print

        CASE    #1      ;print last only
        MOV     #m2,R0
        JSR     @print
        BREAK

        CASE    #2      ;print first only
        MOV     #m1,R0
        JSR     @print
        BREAK

        DEFAULT
        MOV     #1,R1L  ;print both
        CONTINUE        ;go to the SWITCH

        ENDS
        RTS

m1      DC.B    "This is message one."
m2      DC.B    "And this is message two."

        END
```

This produces the following code:

```
1    00000000                NAME   break
2    00000000                EXTERN  print
3    00000000                ASEG    0
4    00000000 0064           DC.W    start   ;reset vector
5    00000002
6    00000064                ORG     100
7    00000064 F901    start  MOV     #1,R1L
8    00000066 5E00006C       JSR     @print_msg
9    0000006A 40FE           BRA     $       ;wait forever
```

```
10    0000006C
11    0000006C          print_msg
12    0000006C                    SWITCH  R1L
13    0000006C
14    0000006C                    CASE    #0      ;print both messages
14.1  0000006C A900     _?0       CMP #0,R1L
14.2  0000006E 4608               BNE _?2
15    00000070 7900009A           MOV     #m1,R0
16    00000074 5E......           JSR     @print
17    00000078
18    00000078                    CASE    #1      ;print last only
18.1  00000078 A901     _?2       CMP #1,R1L
18.2  0000007A 460A               BNE _?3
19    0000007C 790000AF           MOV     #m2,R0
20    00000080 5E......           JSR     @print
21    00000084                    BREAK
21.1  00000084 4012               BRA _?1
22    00000086
23    00000086                    CASE    #2      ;print first only
23.1  00000086 A902     _?3       CMP #2,R1L
23.2  00000088 460A               BNE _?4
24    0000008A 7900009A           MOV     #m1,R0
25    0000008E 5E......           JSR     @print
26    00000092                    BREAK
26.1  00000092 4004               BRA _?1
27    00000094
28    00000094                    DEFAULT
28.1  00000094          _?4
29    00000094 F901               MOV     #1,R1L  ;print both
30    00000096                    CONTINUE        ;go to the SWITCH
30.1  00000096 40D4               BRA _?0
31    00000098
32    00000098                    ENDS
32.1  00000098          _?1
33    00000098 5470               RTS
34    0000009A
35    0000009A 54686973 m1        DC.B    "This is message one."
36    000000AF 416E6420 m2        DC.B    "And this is message two."
37    000000C8
38    000000C8                    END
```

LISTING CONTROL DIRECTIVES

These directives provide control over the assembler listing.

Directive	*Description*
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.
LSTOUT	Controls assembly listing output.
LSTPAG	Controls the formatting of output into pages.
LSTREP	Controls the listing of lines generated by repeat directives.
LSTSAS	Controls structured assembly listing.
LSTXRF	Generates a cross reference table.
LSTCYC	Controls the listing of cycle counts.
CYCLES	Sets the cycle count.
PAGSIZ	Sets the number of lines per page.
COL	Sets the number of columns per page.
PAGE	Generates a new page.

The following directives are provided for backward compatibility only, and are ignored:

LSTFOR, LSTWID, TITL, STITL, PTITL, and PSTITL.

SYNTAX

```
LSTCND{+ | -}
LSTCOD{+ | -}
LSTEXP{+ | -}
LSTMAC{+ | -}
LSTOUT{+ | -}
LSTPAG{+ | -}
```

```
LSTREP{+ | -}
LSTSAS{+ | -}
LSTXRF{+ | -}
LSTCYC{+ | -}
CYCLES count
COL columns
PAGSIZ lines
PAGE
```

PARAMETERS

count	Value to which the cycle count is set.
columns	An absolute expression in the range 80 to 132, default 132.
lines	An absolute expression in the range 10 to 150.

DESCRIPTION

Turning the listing on or off
Use LSTOUT- to disable all list output except for error messages. This overrides all other list 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 expand the listing of output code to more than one line if needed; ie long ASCII strings will produce several lines of list output.

The default setting is LSTCOD-, which lists only the first line of code for a source line; 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 REPT, REPTC, and REPTI directives.

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

Controlling structured assembly listing
Use LSTSAS- to disable listing of the assembly source produced by the directives for structured assembly.

The default is LSTSAS+, which lists assembly source produced by structured assembly directives.

Generating a cross reference table
Use LSTXRF+ to generate a cross reference table at the end of the assembly 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.

Generating cycle counts
Use LSTCYC+ to list cycle counts. The value displayed is the sum of processor clock cycles, and the sum can be reset to any value by the CYCLES directive. The cycle count is set to 0 at the beginning of the listing.

Note that the cycle count shown assumes no caches or cycle overlap or other runtime dependencies.

Formatting listed output
Use COL to set the number of columns per page of the assembly list. The default number of columns is 132.

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

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

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

Use PAGE to generate a new page in the assembly listing 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:

```
 1    00000000                 NAME    lstcnd
 2    00000000                 EXTERN  print
 3    00000000
 4    00000000                 RSEG    prom
 5    00000000          debug  SET     0
 6    00000000
 7    00000000          begin  IF      debug
 8    00000000                 JSR     print
 9    00000000                 ENDIF
10    00000000
11    00000000                 LSTCND+
12    00000000          begin2 IF      debug
14    00000000                 ENDIF
15    00000000                 END     begin
```

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

```
 1    00000000                 NAME    lstcod
 2    00000000
 3    00000000 0001000A table   DC      1,10,100,1000,10000
 4    0000000A
 5    0000000A                 LSTCOD+
 6    0000000A 0001000A table2  DC      1,10,100,1000,10000
               006403E8
               2710
 7    00000014                 END
```

Controlling the listing of macros

The following example shows the effect of LSTMAC and LSTEXP:

```
        NAME    lstmac

times2  MACRO   reg
        SHAL    reg
        ENDM

        LSTMAC+
div2    MACRO   reg
        SHLR    reg
        ENDM

begin   times2  R2L

        LSTEXP-
        div2    R1H
        RTS

        END     begin
```

This will produce the following output:

```
     1    00000000                NAME    lstmac
     2    00000000
     6    00000000
     7    00000000                LSTMAC+
     8    00000000         div2   MACRO   reg
     9    00000000                SHLR    reg
    10    00000000                ENDM
    11    00000000
    12    00000000         begin  times2  R2L
  12.1    00000000 108A           SHAL    R2L
  12.2    00000002                ENDM
    13    00000002
    14    00000002                LSTEXP-
    15    00000002                div2    R1H
    16    00000004 5470           RTS
    17    00000006
    18    00000006                END     begin
```

Controlling the listing of generated lines

The following example illustrates the effect of LSTREP-:

```
    1    00000000                NAME    tables
    2    00000000
    3    00000000          main  ; Generate table of powers of 3
    4    00000001          calc  SET     1
    5    00000000                REPT    4
    6    00000000                DC.W    calc
    7    00000000          calc  SET     calc * 3
    8    00000000                ENDR
  8.1    00000000 0001           DC.W    calc
  8.2    00000003          calc  SET     calc * 3
  8.3    00000002 0003           DC.W    calc
  8.4    00000009          calc  SET     calc * 3
  8.5    00000004 0009           DC.W    calc
  8.6    0000001B          calc  SET     calc * 3
  8.7    00000006 001B           DC.W    calc
  8.8    00000051          calc  SET     calc * 3
    9    00000008
   10    00000008                LSTREP-
   11    00000008                ; Generate table of powers of 7
   12    00000001          calc  SET     1
   13    00000008                REPT    4
   14    00000008                DC.W    calc
   15    00000008          calc  SET     calc * 7
   16    00000008                ENDR
   17    00000010
   18    00000010                END     main
```

Controlling structured assembly listing

The following example illustrates the effect of LSTSAS-:

```
        NAME    lstsas

begin   IFS     R1H <LT> #7 THEN
        ROTL    R1H
        ENDIFS

        LSTSAS-
        IFS     R1H <LT> #7 THEN
        ROTL    R1H
        ENDIFS

        END     begin
```

This will generate the following listing:

```
 1    00000000                  NAME    lstsas
 2    00000000
 3    00000000          begin   IFS     R1H <LT> #7 THEN
3.1   00000000 A107            CMP #7,R1H
3.2   00000002 4C02            BGE _?0
 4    00000004 1281            ROTL    R1H
 5    00000006                  ENDIFS
5.1   00000006          _?0
 6    00000006
 7    00000006                  LSTSAS-
 8    00000006                  IFS     R1H <LT> #7 THEN
 9    0000000A 1281            ROTL    R1H
10    0000000C                  ENDIFS
11    0000000C
12    0000000C                  END     begin
```

Generating a cross reference table

The following listing shows the cross reference table generated by LSTXRF+:

```
 1    00000000          /* Euclids's Algorithm */
 2    00000000          /* Numbers in R0L and R1L */
 3    00000000          /* Result is in R1L */
 4    00000000
 5    00000000                  NAME    GCD
 6    00000000                  LSTXRF+
 7    00000000
 8    00000000          begin   REPEAT
8.1   00000000          _?0
 9    00000000 F100            MOV     #0,R1H
10    00000002 5181            DIVXU   R0L,R1  ;Remainder = R1H
11    00000004 0C19            MOV     R1H,R1L
12    00000006                  IFS     R0L <GT> R1L THEN
12.1  00000006 1C98            CMP R1L,R0L
12.2  00000008 4F04            BLE _?2
13    0000000A 0C89            MOV     R0L,R1L
14    0000000C 0C18            MOV     R1H,R0L
15    0000000E                  ENDIFS
15.1  0000000E          _?2
16    0000000E                  UNTIL   R0L <EQ> #0
16.1  0000000E A800            CMP #0,R0L
```

```
    16.2  00000010 46EE             BNE _?0
    16.3  00000012           _?1
    17    00000012 5470             RTS     ; Result in R1L
    18    00000014
    19    00000014                  END     begin


Segment             Type       Mode
----------------------------------------
ASEG                CODE       ABS Org:0

Label               Mode   Type                    Segment    Value
------------------------------------------------------------------------------
BEGIN               ABS    CONST UNTYP.            ASEG       0
_?0                 ABS    CONST UNTYP.            ASEG       0
_?1                 ABS    CONST UNTYP.            ASEG       12
_?2                 ABS    CONST UNTYP.            ASEG       E
```

Generating cycle counts

The following example uses `CYCLES 0` to calculate the cycle count for the main loop in a capture routine:

```
  1           00000000                  NAME    cycles
  2         0 00000000                  LSTCYC+
  3         0 00000000                  EXTERN  ioport
  4         0 00000000
  5         0 00000000                  RSEG    ram
  6         0 00000000          inbuf   DS.B    256
  7         0 00000100
  8         0 00000000                  RSEG    prom
  9         0 00000000 79000100 capture MOV     #256,R0
 10         0 00000004                  CYCLES  0
 11         0 00000004 6A0B.... capt2   MOV     @ioport,R3L
 12         6 00000008 6C8B             MOV     R3L,@-R0
 13        12 0000000A A800             CMP     #0,R0L
 14        16 0000000C 46F6             BNE     capt2
 15        20 0000000E 5470             RTS
 16        30 00000010
 17        30 00000010                  END
```

The total count is 20 cycles, excluding the `RTS` instruction.

Formatting listed output
The following example formats the output into pages of 66 lines each with 80 columns. The `PAGE` directive inserts a page break between modules:

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

C-STYLE PREPROCESSOR DIRECTIVES

The following C-language preprocessor directives are available:

Directive	*Description*
`#define`	Assigns a value to a label.
`#undef`	Undefines a label.
`#if`	Assembles instructions if a condition is true.
`#ifdef`	Assembles instructions if a symbol is defined.
`#ifndef`	Assembles instructions if a symbol is undefined.
`#else`	Assembles instructions if a condition is false.
`#endif`	Ends a `#if`, `#ifdef`, or `#ifndef` block.
`#include`	Includes a file.
`#error`	Generates an error.
`#pragma`	Ignores the rest of the line.

SYNTAX

`#define` *`label text`*

`#undef` *`label`*

`#if` *`condition`*

`#ifdef` *`label`*

```
#ifndef label
#else
#endif
#include {"filename" | <filename>}
#error "message"
#pragma line
```

PARAMETERS

label	Symbol to be defined, undefined, or tested.	
text	Value to be assigned.	
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.
filename	Name of file to be included.	
message	Text to be displayed.	
line	Text ignored.	

DESCRIPTION

Defining and undefining labels

Use #define to define a temporary label.

```
#define label value
```

is similar to:

```
label SET value
```

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

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 (ie 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.

For example, to include the assembler source file macros.s37 you might specify:

```
#include "c:\iar\asm\inc\macros.s37"
```

You can use the AH8_INC environment variable to specify the path to the include directory. For example, if you include the following line in the autoexec.bat file:

```
AH8_INC=c:\iar\asm\inc\
```

you can shorten the above #include statement to:

```
#include "macros.s37"
```

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

Ignoring #pragmas

A `#pragma` line is ignored, making it easier to have header files common to C and assembler.

EXAMPLES

Using conditional directives

The following example defines a label `adjust`, and then uses the conditional directive `#ifdef` to use the value if it is defined:

```
        EXTERN  input
#define adjust  10

main    MOV     @input,R0L
#ifdef  adjust
        ADD.B   #adjust,R0L
#else
        ADD.B   #7,R0L
#endif
        MOV     R0L,@input
        RTS

        END
```

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.s37`:

```
subi    MACRO   const,reg
        ORC     #H'05,CCR
        SUBX    #(const-1),reg
        ENDM

addi    MACRO   const,reg
        ADD.B   #const,reg
        ENDM
```

The macro definitions can then be included as in the following example:

```
        NAME    include

; Standard macro definitions
#include "macros.s37"

; Program
main    MOV     #123,R0L
```

```
        subi    99,R0L
        RTS

        END     main
```

Displaying errors
The following example generates an error if a label is undefined:

```
main    MOV     #3,R0
#ifndef level
#error  "Not defined"
#endif
        RTS

        END main
```

DATA DEFINITION OR ALLOCATION DIRECTIVES

These directives define temporary values or reserve memory.

Directive	*Description*
DS	Allocates space.
DC	Generates constants.

SYNTAX

DS[.*size*] *items*

DC[.*size*] *expr* [,*expr*] …

PARAMETERS

size The size of generated values; one of:

- .B Byte
- .W Word (the default); ie 2 bytes.
- .L Long word; ie 4 bytes.
- .S Single; ie 4 byte floating-point constant.
- .D Double; ie 8 byte floating-point constant.

items An absolute expression specifying the number of items to be reserved.

expr A valid absolute, relocatable, or external expression, or an ASCII string. ASCII strings will be zero filled to a multiple of *size*.

DESCRIPTION

Use DS to allocate space. The memory contents are not initialized in any way.

Use DC to initialize and reserve memory space.

EXAMPLES

Reserving space
To reserve space for 0xA words:

```
table   DS      0xA
```

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

```
        NAME    table

table   DC      addsubr,subsubr,clrsubr

addsubr ADD     R1,R0
        RTS

subsubr SUB     R1,R0
        RTS

clrsubr MOV     #0,R0
        RTS

        END
```

Defining strings
To define a string:

```
mymess  DC.B    'Please enter your name'
```

To define a string which includes a trailing zero:

```
myCstr  DC.B    "This is a string."
```

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

```
errmess DC.B    'Don''t understand!'
```

ASSEMBLER CONTROL DIRECTIVES

These directives provide control over the operation of the assembler.

Directive	*Description*
`$`	Includes a file.
`/*comment*/`	C-style comment delimiter.
`RADIX`	Sets the default base.
`CASEON`	Enables case sensitivity.
`CASEOFF`	Disables case sensitivity.
`OPT`	Sets assembler options.
`MODEL`	Specifies the memory model.

SYNTAX

`$`*`filename`*

`/*comment*/`

`RADIX` *`expr`*

`CASEON`

`CASEOFF`

`OPT` *`option`* `[,`*`option`*`] …`

`MODEL` *`model`*

PARAMETERS

`filename`	Name of file to be included. The `$` character must be the first character on the line.
`comment`	Comment ignored by the assembler.
`expr`	Default base; default 10 (decimal).
`option`	One or more of the options shown in the table below.
`model`	Memory model, one of 0, 1, or 2, which must be resolvable.

DESCRIPTION

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

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

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

To reset the base from 16 to 10 *`expr`* must be written in hexadecimal. For example:

```
RADIX 0x0A
```

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.

Setting assembler options
Use `OPT` to set default size (number of bits) in PC-relative branches and displacements in register-indirect-with-displacement operands. `OPT` can take the following arguments:

Option	*Mnemonic*	*Description*
`BRB`	BRanch Byte.	Sets default branch size to byte (8-bits). This is the default when `BRON` is used for the first time.
`BRW`	BRanch Word.	Sets default branch size to word (16-bits).
`BRON`	BRanch size ON.	Activates the selected default size (`BRB` or `BRW`). The `BRB` and `BRW` directives have no effect until this directive is encountered.
`BROFF`	BRanch size OFF.	Deactivates the selected default size (`BRB` or `BRW`). This is the default at the start of assembly.

Option	*Mnemonic*	*Description*
RIW	Register Indirect displacement Word.	Sets default displacement size to word (16-bits). This is the default when RION is used for the first time.
RIL	Register Indirect displacement Long.	Sets default displacement size to long (32-bits). Can only be used for H8 family and replaces the RITB option as the H8 chips cannot have a 24-bit displacement.
RITB	Register Indirect displacement Three Bytes.	Sets default displacement size to 3 bytes word (24-bits). This option only works for the H8/300H.
RION	Register Indirect displacement ON.	Analagous to BRON.
RIOFF	Register Indirect displacement OFF.	Analagous to BROFF. This is the default at the start of assembly.

Operand sizes expressed with :8, :16, :24, :32 etc override the defaults selected with the OPT directive.

Specifying the memory model
Use MODEL to inform the assembler about the memory model used.

MODEL is used to determine the valid ranges for the addressing modes @aa:8 and @aa:16. The alternative values for MODEL and the corresponding address ranges are as follows:

Model	*@aa:8*	*@aa:16*	*Addressable*
0	FF00–FFFF	0–FFFF	64 Kbyte
1	FFFF00–FFFFFF	0–7FFF, FF8000–FFFFFF	16 Mbyte
2	FFFFFF00–FFFFFFFF	0–7FFF, FFFF8000–FFFFFFFF	4 Gbyte

The command line option `-vn`, where `n` is 0 to 2, and memory model option `-m` correspond to the chip and `MODEL` values as follows:

Option	*Chip*	*Addressable area*	*Model*
`-v0 -ms`	H8/300H	64 Kbyte	0
`-v0 -ml`	H8/300H	16 Mbyte	1
`-v1 -ms`	H8S/2200	64 Kbyte	0
`-v1 -ml`	H8S/2200	4 Gbyte	2
`-v2 -ms`	H8S/2600	64 Kbyte	0
`-v2 -ml`	H8S/2600	16 Mbyte	1

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 `macros.s37`:

```
subi    MACRO   const,reg
        ORC     #H'05,CCR
        SUBX    #(const-1),reg
        ENDM

addi    MACRO   const,reg
        ADD.B   #const,reg

        ENDM
```

The macro definitions can be included as in the following example:

```
        NAME    include

; Standard macro definitions
$macros.s37

; Program
main    MOV     #123,R0L
        subi    99,R0L
        RTS

        END     main
```

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

```
/*
Program to read serial input.
Version 2: 19.6.94
Author: mjp
*/
```

Changing the base
To set the default base to `16`:

```
        RADIX   16
        MOV     #12,R3
```

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

Controlling case sensitivity
By default `CASEOFF` is active, so in the following example `label` and `LABEL` are identical:

```
label   NOP     ;stored as "LABEL"
        JMP     LABEL
```

However, the following will generate a duplicate label error:

```
label   NOP
        CASEON
LABEL   NOP     ;Error: "LABEL" already defined
        END
```

Setting assembler options
To set the default branch size to word:

```
OPT BRW,BRON
```

ASSEMBLER INSTRUCTIONS

This chapter lists the H8 Series instruction mnemonics.

INTRODUCTION

The following symbols are used in the list of instruction mnemonics:

Symbol	*What it means*
`#xx:3`	Immediate 3-bit data.
`#xx:8`	Immediate 8-bit data.
`#xx:16`	Immediate 16-bit data.
`#xx:32`	Immediate 32-bit data.
`dp:8`	8-bit displacement.
`d:16`	16-bit displacement.
`Rs,ERs`	Source register.
`Rd,ERd`	Destination register.
`@Rd`	Register indirect.
`@(d:16,Rd)`	Register indirect with 16-bit displacement.
`@(d:24,Rd)`	Register indirect with 24-bit displacement.
`@(d:32,Rd)`	Register indirect with 32-bit displacement.
`@Rd+`	Register indirect with post-increment.
`@-Rd`	Register indirect with pre-decrement.
`@aa:8`	Absolute 8-bit address.
`@aa:16`	Absolute 16-bit address.
`@aa:24`	Absolute 24-bit address.
`@aa:32`	Absolute 32-bit address.
`@(d:8,PC)`	PC-relative, 8-bit displacement.
`@@aa:8`	Memory indirect.

Symbol	*What it means*
PC	Program counter.
SP	Stack pointer*.
CCR	Condition code register.
EXR	Extended control register.
MACL, MACH	Multiply-accumulate register.
C	Carry flag.
V	Overflow flag
Z	Zero flag.
N	Negative flag.
H	Half-carry flag.
I	Interrupt mask.
†	Not available on the H8/300H.
§	Not available on the H8S.

* Note that the alias SP (stack pointer) is provided for assembler files generated by the H8 C Compiler, and it is recommended that ER7, R7, or R7L should be used in preference.

ADD

Add binary.

Addressing mode	*Syntax*		*Size*
Immediate	ADD.B	#xx:8,Rd	B
Register direct	ADD.B	Rs,Rd	B
Immediate	ADD.W	#xx:16,Rd	W
Register direct	ADD.W	Rs,Rd	W
Immediate	ADD.L	#xx:32,ERd	L
Register direct	ADD.L	ERs,ERd	L

ADDS

Add immediate with sign extension.

Addressing mode	*Syntax*		*Size*
Register direct	ADDS	#1,ERd	L
Register direct	ADDS	#2,ERd	L
Register direct	ADDS	#4,ERd	L

ADDX

Add with extend carry.

Addressing mode	*Syntax*		*Size*
Immediate	ADDX.B	#xx:8,Rd	B
Register direct	ADDX.B	Rs,Rd	B

AND

AND logical.

Addressing mode	*Syntax*		*Size*
Immediate	AND.B	#xx:8,Rd	B
Register direct	AND.B	Rs,Rd	B
Immediate	AND.W	#xx:16,Rd	W
Register direct	AND.W	Rs,Rd	W
Immediate	AND.L	#xx:32,ERd	L
Register direct	AND.L	ERs,ERd	L

ANDC

AND control register.

Addressing mode		*Syntax*		*Size*
Immediate		ANDC	#xx:8,CCR	B
Immediate	†	ANDC	#xx:8,EXR	B

BAND

Bit AND.

Addressing mode		*Syntax*		*Size*
Register direct		BAND	#xx:3,Rd	B
Register indirect		BAND	#xx:3,@ERd	B
Absolute address		BAND	#xx:3,@aa:8	B
Absolute address	†	BAND	#xx:3,@aa:16	B
Absolute address	†	BAND	#xx:3,@aa:32	B

BCC (BHS)

Branch if carry clear (high or same).

Addressing mode	*Syntax*		*Size*
PC relative	BCC(BHS)	d:8	–
PC relative	BCC(BHS)	d:16	–

BCLR

Bit clear.

Addressing mode		*Syntax*		*Size*
Register direct		BCLR	#xx:3,Rd	B
Register indirect		BCLR	#xx:3,@ERd	B
Absolute address		BCLR	#xx:3,@aa:8	B
Absolute address	†	BCLR	#xx:3,@aa:16	B
Absolute address	†	BCLR	#xx:3,@aa:32	B
Register direct		BCLR	Rn,Rd	B
Register indirect		BCLR	Rn,@ERd	B
Absolute address		BCLR	Rn,@aa:8	B
Absolute address	†	BCLR	Rn,@aa:16	B
Absolute address	†	BCLR	Rn,@aa:32	B

BCS (BLO)

Branch if carry set (low).

Addressing mode	*Syntax*		*Size*
PC relative	`BCS(BLO)`	`d:8`	–
PC relative	`BCS(BLO)`	`d:16`	–

BEQ

Branch if equal.

Addressing mode	*Syntax*		*Size*
PC relative	`BEQ`	`d:8`	–
PC relative	`BEQ`	`d:16`	–

BF (BRN)

Branch if false.

Addressing mode	*Syntax*		*Size*
PC relative	`BF(BRN)`	`d:8`	–
PC relative	`BF(BRN)`	`d:16`	–

BGE

Branch if greater or equal.

Addressing mode	*Syntax*		*Size*
PC relative	`BGE`	`d:8`	–
PC relative	`BGE`	`d:16`	–

BGT

Branch if greater than.

Addressing mode	*Syntax*		*Size*
PC relative	`BGT`	`d:8`	–
PC relative	`BGT`	`d:16`	–

BHI

Branch if high.

Addressing mode		*Syntax*		*Size*
PC relative		BHI	d:8	–
PC relative		BHI	d:16	–

BIAND

Bit invert AND.

Addressing mode		*Syntax*		*Size*
Register direct		BIAND	#xx:3,Rd	B
Register indirect		BIAND	#xx:3,@ERd	B
Absolute address		BIAND	#xx:3,@aa:8	B
Absolute address	†	BIAND	#xx:3,@aa:16	B
Absolute address	†	BIAND	#xx:3,@aa:32	B

BILD

Bit invert load.

Addressing mode		*Syntax*		*Size*
Register direct		BILD	#xx:3,Rd	B
Register indirect		BILD	#xx:3,@ERd	B
Absolute address		BILD	#xx:3,@aa:8	B
Absolute address	†	BILD	#xx:3,@aa:16	B
Absolute address	†	BILD	#xx:3,@aa:32	B

BIOR

Bit invert OR.

Addressing mode		*Syntax*		*Size*
Register direct		BIOR	#xx:3,Rd	B
Register indirect		BIOR	#xx:3,@ERd	B
Absolute address		BIOR	#xx:3,@aa:8	B
Absolute address	†	BIOR	#xx:3,@aa:16	B
Absolute address	†	BIOR	#xx:3,@aa:32	B

BIST

Bit invert store.

Addressing mode		*Syntax*		*Size*
Register direct		BIST	#xx:3,Rd	B
Register indirect		BIST	#xx:3,@ERd	B
Absolute address		BIST	#xx:3,@aa:8	B
Absolute address	†	BIST	#xx:3,@aa:16	B
Absolute address	†	BIST	#xx:3,@aa:32	B

BIXOR

Bit invert exclusive OR.

Addressing mode		*Syntax*		*Size*
Register direct		BIXOR	#xx:3,Rd	B
Register indirect		BIXOR	#xx:3,@ERd	B
Absolute address		BIXOR	#xx:3,@aa:8	B
Absolute address	†	BIXOR	#xx:3,@aa:16	B
Absolute address	†	BIXOR	#xx:3,@aa:32	B

BLD

Bit load.

Addressing mode		*Syntax*		*Size*
Register direct		BLD	#xx:3,Rd	B
Register indirect		BLD	#xx:3,@ERd	B
Absolute address		BLD	#xx:3,@aa:8	B
Absolute address	†	BLD	#xx:3,@aa:16	B
Absolute address	†	BLD	#xx:3,@aa:32	B

BLE

Branch if less or equal.

Addressing mode	*Syntax*		*Size*
PC relative	BLE	d:8	–
PC relative	BLE	d:16	–

BLS

Branch if low or same.

Addressing mode	*Syntax*		*Size*
PC relative	BLS	d:8	–
PC relative	BLS	d:16	–

BLT

Branch if less than.

Addressing mode	*Syntax*		*Size*
PC relative	BLT	d:8	–
PC relative	BLT	d:16	–

BMI

Branch if minus.

Addressing mode	*Syntax*		*Size*
PC relative	BMI	d:8	–
PC relative	BMI	d:16	–

BNE

Branch if not equal.

Addressing mode	*Syntax*		*Size*
PC relative	BNE	d:8	–
PC relative	BNE	d:16	–

BNOT

Bit NOT.

Addressing mode		*Syntax*		*Size*
Register direct		BNOT	#xx:3,Rd	B
Register indirect		BNOT	#xx:3,@ERd	B
Absolute address		BNOT	#xx:3,@aa:8	B
Absolute address	†	BNOT	#xx:3,@aa:16	B
Absolute address	†	BNOT	#xx:3,@aa:32	B
Register direct		BNOT	Rn,Rd	B
Register indirect		BNOT	Rn,@ERd	B
Absolute address		BNOT	Rn,@aa:8	B
Absolute address	†	BNOT	Rn,@aa:16	B
Absolute address	†	BNOT	Rn,@aa:32	B

BOR

Bit OR.

Addressing mode		*Syntax*		*Size*
Register direct		BOR	#xx:3,Rd	B
Register indirect		BOR	#xx:3,@ERd	B
Absolute address		BOR	#xx:3,@aa:8	B
Absolute address	†	BOR	#xx:3,@aa:16	B
Absolute address	†	BOR	#xx:3,@aa:32	B

BPL

Branch if plus.

Addressing mode	*Syntax*		*Size*
PC relative	BPL	d:8	–
PC relative	BPL	d:16	–

BRA (BT)

Branch always.

Addressing mode	*Syntax*		*Size*
PC relative	BRA(BT)	d:8	–
PC relative	BRA(BT)	d:16	–

BRN (BF)

Branch never.

Addressing mode	*Syntax*		*Size*
PC relative	BRN(BF)	d:8	–
PC relative	BRN(BF)	d:16	–

BSET

Bit set.

Addressing mode		*Syntax*		*Size*
Register direct		BSET	#xx:3,Rd	B
Register indirect		BSET	#xx:3,@ERd	B
Absolute address		BSET	#xx:3,@aa:8	B
Absolute address	†	BSET	#xx:3,@aa:16	B
Absolute address	†	BSET	#xx:3,@aa:32	B
Register direct		BSET	Rn,Rd	B
Register indirect		BSET	Rn,@ERd	B
Absolute address		BSET	Rn,@aa:8	B
Absolute address	†	BSET	Rn,@aa:16	B
Absolute address	†	BSET	Rn,@aa:32	B

BSR

Branch to subroutine.

Addressing mode	*Syntax*		*Size*
PC relative	BSR	d:8	–
PC relative	BSR	d:16	–

BST

Bit store.

Addressing mode		*Syntax*		*Size*
Register direct		BST	#xx:3,Rd	B
Register indirect		BST	#xx:3,@ERd	B
Absolute address		BST	#xx:3,@aa:8	B
Absolute address	†	BST	#xx:3,@aa:16	B
Absolute address	†	BST	#xx:3,@aa:32	B

BT (BRA)

Branch if true.

Addressing mode		*Syntax*		*Size*
PC relative		BT(BRA)	d:8	–
PC relative		BT(BRA)	d:16	–

BTST

Bit test.

Addressing mode		*Syntax*		*Size*
Register direct		BTST	#xx:3,Rd	B
Register indirect		BTST	#xx:3,@ERd	B
Absolute address		BTST	#xx:3,@aa:8	B
Absolute address	†	BTST	#xx:3,@aa:16	B
Absolute address	†	BTST	#xx:3,@aa:32	B
Register direct		BTST	Rn,Rd	B
Register indirect		BTST	Rn,@ERd	B
Absolute address		BTST	Rn,@aa:8	B
Absolute address	†	BTST	Rn,@aa:16	B
Absolute address	†	BTST	Rn,@aa:32	B

BVC

Branch if overflow clear.

Addressing mode	*Syntax*		*Size*
PC relative	BVC	d:8	–
PC relative	BVC	d:16	–

BVS

Branch if overflow set.

Addressing mode		*Syntax*		*Size*
PC relative		BVS	d:8	–
PC relative		BVS	d:16	–

BXOR

Bit exclusive OR.

Addressing mode		*Syntax*		*Size*
Register direct		BXOR	#xx:3,Rd	B
Register indirect		BXOR	#xx:3,@ERd	B
Absolute address		BXOR	#xx:3,@aa:8	B
Absolute address	†	BXOR	#xx:3,@aa:16	B
Absolute address	†	BXOR	#xx:3,@aa:32	B

CLRMAC

Clear MAC registers.

Addressing mode		*Syntax*		*Size*
–	†	CLRMAC		–

CMP

Compare.

Addressing mode		*Syntax*		*Size*
Immediate		CMP.B	#xx:8,Rd	B
Register direct		CMP.B	Rs,Rd	B
Immediate		CMP.W	#xx:16,Rd	W
Register direct		CMP.W	Rs,Rd	W
Immediate		CMP.L	#xx:32,ERd	L
Register direct		CMP.L	ERs,ERd	L

DAA

Decimal adjust add.

Addressing mode	*Syntax*		*Size*
Register direct	DAA	Rd	B

DAS

Decimal adjust subtract.

Addressing mode	*Syntax*		*Size*
Register direct	DAS	Rd	B

DEC

Decrement.

Addressing mode	*Syntax*		*Size*
Register direct	DEC.B	Rd	B
Immediate	DEC.W	#1,Rd	W
Immediate	DEC.W	#2,Rd	W
Immediate	DEC.L	#1,ERd	L
Immediate	DEC.L	#2,ERd	L

DIVXS

Divide extend as signed.

Addressing mode	*Syntax*		*Size*
Register direct	DIVXS.B	Rs,Rd	B
Register direct	DIVXS.W	Rs,ERd	W

DIVXU

Divide extend as unsigned.

Addressing mode	*Syntax*		*Size*
Register direct	`DIVXU.B`	`Rs,Rd`	`B`
Register direct	`DIVXU.W`	`Rs,ERd`	`W`

EEPMOV

Move data to EEPROM.

Addressing mode	*Syntax*	*Size*
–	`EEPMOV.B`	`B`
–	`EEPMOV.W`	`W`

EXTS

Zero extend signed.

Addressing mode	*Syntax*		*Size*
Register direct	`EXTS.W`	`Rd`	`W`
Register direct	`EXTS.L`	`ERd`	`L`

EXTU

Zero extend unsigned.

Addressing mode	*Syntax*		*Size*
Register direct	`EXTU.W`	`Rd`	`W`
Register direct	`EXTU.L`	`ERd`	`L`

INC

Increment.

Addressing mode		*Syntax*		*Size*
Register direct		INC.B	Rd	B
Immediate		INC.W	#1,Rd	W
Immediate		INC.W	#2,Rd	W
Immediate		INC.L	#1,ERd	L
Immediate		INC.L	#2,ERd	L

JMP

Jump.

Addressing mode		*Syntax*		*Size*
Register indirect	§	JMP	@Rn	–
Register indirect		JMP	@ERn	–
Absolute address	§	JMP	@aa:16	–
Absolute address		JMP	@aa:24	–
Memory indirect		JMP	@@aa:8	–

JSR

Jump to subroutine.

Addressing mode		*Syntax*		*Size*
Register indirect	§	JSR	@Rn	–
Register indirect		JSR	@ERn	–
Absolute address	§	JSR	@aa:16	–
Absolute address		JSR	@aa:24	–
Memory indirect		JSR	@@aa:8	–

LDC

Load control register.

Addressing mode		*Syntax*		*Size*
Immediate		LDC	#xx:8,CCR	B
Immediate	†	LDC	#xx:8,EXR	B
Register direct		LDC	Rs,CCR	B
Register direct	†	LDC	Rs,EXR	B
Register indirect		LDC	@ERs,CCR	W
Register indirect	†	LDC	@ERs,EXR	W
Register indirect with displacement		LDC	@(d:16,ERs),CCR	W
Register indirect with displacement	†	LDC	@(d:16,ERs),EXR	W
Register indirect with displacement	§	LDC	@(d:24,ERs),CCR	W
Register indirect with displacement	†	LDC	@(d:32,ERs),CCR	W
Register indirect with displacement	†	LDC	@(d:32,ERs),EXR	W
Register indirect with post-increment		LDC	ERs+,CCR	W
Register indirect with post-increment	†	LDC	ERs+,EXR	W
Absolute address		LDC	@aa:16,CCR	W
Absolute address	†	LDC	@aa:16,EXR	W
Absolute address	§	LDC	@aa:24,CCR	W
Absolute address	†	LDC	@aa:32,CCR	W
Absolute address	†	LDC	@aa:32,EXR	W

LDM

Load multiple registers.

Addressing mode		*Syntax*		*Size*
–	†	LDM.L	@SP+,(ER_n-ER_{n+1})	L
–	†	LDM.L	@SP+,(ER_n-ER_{n+2})	L
–	†	LDM.L	@SP+,(ER_n-ER_{n+3})	L

LDMAC

Load to MAC register.

Addressing mode		*Syntax*		*Size*
–	†	LDMAC	ERs,MACH	L
–	†	LDMAC	ERs,MACL	L

MAC

Multiply and accumulate.

Addressing mode		*Syntax*		*Size*
–	†	MAC	@ERn+,@ERm+	–

MOV

Move data.

Addressing mode		*Syntax*		*Size*
Immediate		MOV.B	#xx:8,Rd	B
Register direct		MOV.B	Rs,Rd	B
Register indirect		MOV.B	@ERs,Rd	B
Register indirect with displacement		MOV.B	@(d:16,ERs),Rd	B
Register indirect with displacement	§	MOV.B	@(d:24,ERs),Rd	B
Register indirect with displacement	†	MOV.B	@(d:32,ERs),Rd	B

Addressing mode		*Syntax*		*Size*
Register indirect with post-increment		MOV.B	@ERs+,Rd	B
Absolute address		MOV.B	@aa:8,Rd	B
Absolute address		MOV.B	@aa:16,Rd	B
Absolute address	§	MOV.B	@aa:24,Rd	B
Absolute address	†	MOV.B	@aa:32,Rd	B
Register indirect		MOV.B	Rs,@ERd	B
Register indirect with displacement		MOV.B	Rs,@(d:16,ERd)	B
Register indirect with displacement	§	MOV.B	Rs,@(d:24,ERd)	B
Register indirect with displacement	†	MOV.B	Rs,@(d:32,ERd)	B
Register indirect with pre-decrement		MOV.B	Rs,@-ERd	B
Absolute address		MOV.B	Rs,@aa:8	B
Absolute address		MOV.B	Rs,@aa:16	B
Absolute address		MOV.B	Rs,@aa:24	B
Immediate		MOV.W	#xx:16,Rd	W
Register direct		MOV.W	Rs,Rd	W
Register indirect		MOV.W	@ERs,Rd	W
Register indirect with displacement		MOV.W	@(d:16,ERs),Rd	W
Register indirect with displacement	§	MOV.W	@(d:24,ERs),Rd	W
Register indirect with displacement	†	MOV.W	@(d:32,ERs),Rd	W
Register indirect with post-increment		MOV.W	@ERs+,Rd	W

Addressing mode		*Syntax*		*Size*
Absolute address		MOV.W	@aa:16,Rd	W
Absolute address	§	MOV.W	@aa:24,Rd	W
Absolute address	†	MOV.W	@aa:32,Rd	W
Register indirect		MOV.W	Rs,@ERd	W
Register indirect with displacement		MOV.W	Rs,@(d:16,ERd)	W
Register indirect with displacement	§	MOV.W	Rs,@(d:24,ERd)	W
Register indirect with displacement	†	MOV.W	Rs,@(d:32,ERd)	W
Register indirect with pre-decrement		MOV.W	Rs,@-ERd	W
Absolute address		MOV.W	Rs,@aa:16	W
Absolute address	§	MOV.W	Rs,@aa:24	W
Absolute address	†	MOV.W	Rs,@aa:32	W
Immediate		MOV.L	#xx:32,ERd	L
Register direct		MOV.L	@ERs,ERd	L
Register indirect with displacement		MOV.L	@(d:16,ERs),ERd	L
Register indirect with displacement	§	MOV.L	@(d:24,ERs),ERd	L
Register indirect with displacement	†	MOV.L	@(d:32,ERs),ERd	L
Register indirect with post-increment		MOV.L	@ERs+,ERd	L
Absolute address		MOV.L	Rs,@aa:16,ERd	L
Absolute address	§	MOV.L	Rs,@aa:24,ERd	L
Absolute address	†	MOV.L	Rs,@aa:32,ERd	L
Register indirect		MOV.L	ERs,@ERd	L

Addressing mode		*Syntax*		*Size*
Register indirect with displacement		MOV.L	ERs,@(d:16,ERd)	L
Register indirect with displacement	§	MOV.L	ERs,@(d:24,ERd)	L
Register indirect with displacement	†	MOV.L	ERs,@(d:32,ERd)	L
Register indirect with pre-decrement		MOV.L	ERs,@-ERd	L
Absolute address		MOV.L	ERs,@aa:16	L
Absolute address	§	MOV.L	ERs,@aa:24	L
Absolute address	†	MOV.L	ERs,@aa:32	L

MOVFPE

Move data from peripheral with E clock.

Addressing mode	*Syntax*		*Size*
Absolute address	MOVFPE	@aa:16,Rd	B

MOVTPE

Move data to peripheral with E clock.

Addressing mode	*Syntax*		*Size*
Absolute address	MOVTPE	Rs,@aa:16	B

MULXS

Multiply extend as unsigned.

Addressing mode	*Syntax*		*Size*
Register direct	MULXS.B	Rs,Rd	B
Register direct	MULXS.W	Rs,ERd	W

MULXU

Multiply extend as unsigned.

Addressing mode	*Syntax*		*Size*
Register direct	MULXU.B	Rs,Rd	B
Register direct	MULXU.W	Rs,ERd	W

NEG

Negate.

Addressing mode	*Syntax*		*Size*
Register direct	NEG.B	Rd	B
Register direct	NEG.W	Rd	W
Register direct	NEG.L	ERd	L

NOP

No operation.

Addressing mode	*Syntax*		*Size*
–	NOP		–

NOT

NOT logical.

Addressing mode	*Syntax*		*Size*
Register direct	NOT.B	Rd	B
Register direct	NOT.W	Rd	W
Register direct	NOT.L	ERd	L

OR

OR logical.

Addressing mode		*Syntax*		*Size*
Immediate		OR.B	#xx:8,Rd	B
Register direct		OR.B	Rs,Rd	B
Immediate		OR.W	#xx:16,Rd	W
Register direct		OR.W	Rs,Rd	W
Immediate		OR.L	#xx:32,ERd	L
Register direct		OR.L	ERs,ERd	L

ORC

OR control register.

Addressing mode		*Syntax*		*Size*
Immediate		ORC	#xx:8,CCR	B
Immediate	†	ORC	#xx:8,EXR	B

POP

Pop register from stack.

Addressing mode	*Syntax*		*Size*
Register direct	POP.W	Rd	W
Register direct	POP.L	ERd	L

PUSH

Push register to stack.

Addressing mode	*Syntax*		*Size*
Register direct	PUSH.W	Rd	W
Register direct	PUSH.L	ERd	L

ROTL

Rotate left.

Addressing mode		*Syntax*		*Size*
Register direct		ROTL.B	Rd	B
Register direct	†	ROTL.B	#2,Rd	B
Register direct		ROTL.W	Rd	W
Register direct	†	ROTL.W	#2,Rd	W
Register direct		ROTL.L	ERd	L
Register direct	†	ROTL.L	#2,ERd	L

ROTR

Rotate right.

Addressing mode		*Syntax*		*Size*
Register direct		ROTR.B	Rd	B
Register direct	†	ROTR.B	#2,Rd	B
Register direct		ROTR.W	Rd	W
Register direct	†	ROTR.W	#2,Rd	W
Register direct		ROTR.L	ERd	L
Register direct	†	ROTR.L	#2,ERd	L

ROTXL

Rotate with extend carry left.

Addressing mode		*Syntax*		*Size*
Register direct		ROTXL.B	Rd	B
Register direct	†	ROTXL.B	#2,Rd	B
Register direct		ROTXL.W	Rd	W
Register direct	†	ROTXL.W	#2,Rd	W
Register direct		ROTXL.L	ERd	L
Register direct	†	ROTXL.L	#2,ERd	L

ROTXR

Rotate right.

Addressing mode		*Syntax*		*Size*
Register direct		ROTXR.B	Rd	B
Register direct	†	ROTXR.B	#2,Rd	B
Register direct		ROTXR.W	Rd	W
Register direct	†	ROTXR.W	#2,Rd	W
Register direct		ROTXR.L	ERd	L
Register direct	†	ROTXR.L	#2,ERd	L

RTE

Return from exception.

Addressing mode	*Syntax*	*Size*
–	RTE	–

RTS

Return from subroutine.

Addressing mode	*Syntax*	*Size*
–	RTS	–

SHAL

Shift arithmetic left.

Addressing mode		*Syntax*		*Size*
Register direct		SHAL.B	Rd	B
Register direct	†	SHAL.B	#2,Rd	B
Register direct		SHAL.W	Rd	W
Register direct	†	SHAL.W	#2,Rd	W
Register direct		SHAL.L	ERd	L
Register direct	†	SHAL.L	#2,ERd	L

SHAR

Shift arithmetic right.

Addressing mode		*Syntax*		*Size*
Register direct		SHAR.B	Rd	B
Register direct	†	SHAR.B	#2,Rd	B
Register direct		SHAR.W	Rd	W
Register direct	†	SHAR.W	#2,Rd	W
Register direct		SHAR.L	ERd	L
Register direct	†	SHAR.L	#2,ERd	L

SHLL

Shift logically left.

Addressing mode		*Syntax*		*Size*
Register direct		SHLL.B	Rd	B
Register direct	†	SHLL.B	#2,Rd	B
Register direct		SHLL.W	Rd	W
Register direct	†	SHLL.W	#2,Rd	W
Register direct		SHLL.L	ERd	L
Register direct	†	SHLL.L	#2,ERd	L

SHLR

Shift logically right.

Addressing mode		*Syntax*		*Size*
Register direct		SHLR.B	Rd	B
Register direct	†	SHLR.B	#2,Rd	B
Register direct		SHLR.W	Rd	W
Register direct	†	SHLR.W	#2,Rd	W
Register direct		SHLR.L	ERd	L
Register direct	†	SHLR.L	#2,ERd	L

SLEEP

Sleep.

Addressing mode	*Syntax*	*Size*
–	SLEEP	–

STC

Store control register.

Addressing mode		*Syntax*		*Size*
Register direct		STC	CCR,Rd	B
Register direct	†	STC	EXR,Rd	B
Register indirect		STC	CCR,@ERd	W
Register indirect	†	STC	EXR,@ERd	W
Register indirect with displacement		STC	CCR,@(d:16,ERd)	W
Register indirect with displacement	†	STC	EXR,@(d:16,ERd)	W
Register indirect with displacement	§	STC	CCR,@(d:24,ERd)	W
Register indirect with displacement	†	STC	CCR,@(d:32,ERd)	W
Register indirect with displacement	†	STC	EXR,@(d:32,ERd)	W
Register indirect with pre-decrement		STC	CCR,@-ERd	W
Register indirect with pre-decrement	†	STC	EXR,@-ERd	W
Absolute address		STC	CCR,@aa:16	W
Absolute address	†	STC	EXR,@aa:16	W
Absolute address	§	STC	CCR,@aa:24	W
Absolute address	†	STC	CCR,@aa:32	W
Absolute address	†	STC	EXR,@aa:32	W

STM

Store from multiple registers.

Addressing mode		*Syntax*		*Size*
–	†	STM.L	(ER_n-ER_{n+1}),@-SP	L
–	†	STM.L	(ER_n-ER_{n+1}),@-SP	L
–	†	STM.L	(ER_n-ER_{n+1}),@-SP	L

STMAC

Store from MAC register.

Addressing mode		*Syntax*		*Size*
–	†	STMAC	MACH,ERd	L
–	†	STMAC	MACL,ERd	L

SUB

Subtract binary.

Addressing mode	*Syntax*		*Size*
Register direct	SUB.B	Rs,Rd	B
Immediate	SUB.W	#xx:16,Rd	W
Register direct	SUB.W	Rs,Rd	W
Immediate	SUB.L	#xx:32,ERd	L
Register direct	SUB.L	ERs,ERd	L

SUBS

Subtract immediate with sign extension.

Addressing mode	*Syntax*		*Size*
Register direct	SUBS	#1,ERd	L
Register direct	SUBS	#2,ERd	L
Register direct	SUBS	#4,ERd	L

SUBX

Subtract with extend carry.

Addressing mode	*Syntax*		*Size*
Immediate	SUBX.B	#xx:8,Rd	B
Register direct	SUBX.B	Rs,Rd	B

TAS

Test and set.

Addressing mode		*Syntax*		*Size*
–	†	TAS	@ERd	–

TRAPA

Trap.

Addressing mode	*Syntax*		*Size*
Immediate	TRAPA	#xx:2	–

XOR

Exclusive OR logical.

Addressing mode	*Syntax*		*Size*
Immediate	XOR.B	#xx:8,Rd	B
Register direct	XOR.B	Rs,Rd	B
Immediate	XOR.W	#xx:16,Rd	W
Register direct	XOR.W	Rs,Rd	W
Immediate	XOR.L	#xx:32,ERd	L
Register direct	XOR.L	ERs,ERd	L

XORC

Exclusive OR control register.

Addressing mode		*Syntax*		*Size*
Immediate		`XORC`	`#xx:8,CCR`	`B`
Immediate	†	`XORC`	`#xx:8,EXR`	`B`

XLINK LINKER

This chapter describes the IAR Systems XLINK Linker, and gives examples of how it can be used.

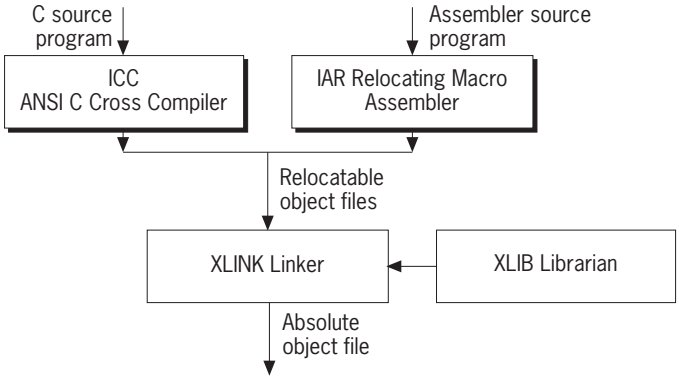
Note that some of the options described in the following chapters may not be available for all assemblers.

INTRODUCTION

The XLINK Linker is a powerful, flexible software tool for use in the development of embedded-controller applications. XLINK reads one or more relocatable object files produced by the IAR Systems Assembler or C Compiler and produces absolute, machine-code programs as output.

It is equally well-suited for linking small, single-file, absolute assembler programs as it is for linking large, relocatable, multi-module, C or mixed C and assembler programs.

The following diagram illustrates the linking process:



OBJECT FORMAT

The object files produced by the IAR Systems Assembler and C Compiler use a proprietary format called UBROF, which stands for Universal Binary Relocatable Object Format. An application can be made up of any number of UBROF relocatable files, in any combination of assembler and C.

XLINK FUNCTIONS

XLINK performs three distinct functions when you link a program:

- It loads modules containing executable code or data from the input file(s).
- It locates each segment of code or data at a user-specified address.
- It links the various modules together by resolving all global (ie non-local, program-wide) symbols that could not be resolved by the assembler or compiler.
- It loads modules needed by the program from user-defined libraries.

LIBRARIES

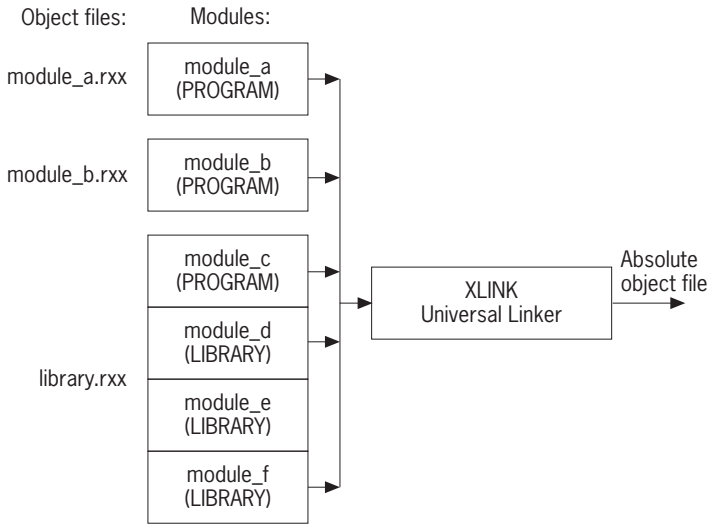
When XLINK reads a library file (which can contain multiple C or assembler modules) it will only load those modules which are actually needed by the program you are linking. This avoids having to load all the modules in a library file when you only need one routine. The XLIB Librarian is used to manage these library files.

OUTPUT FORMAT

The final output produced by XLINK is an absolute, executable object file that can be put into an EPROM, downloaded to a hardware emulator, or executed on the PC using the IAR Systems C-SPY debugger.

INPUT FILES AND MODULES

The following diagram shows how XLINK processes input files and load modules for a typical assembler or C program:



The main program has been assembled from two source files, `module_a.s`*xx* and `module_b.s`*xx*, to produce two relocatable files. Each of these files consists of a single module `module_a` and `module_b`. By default, the assembler assigns the `PROGRAM` attribute to both `module_a` and `module_b`. This means that they will always be loaded and linked whenever the files they are contained in are processed by XLINK; ie the filenames are given as arguments.

The code and data from a single C source file ends up as a single module in the file produced by the compiler. In other words, there is a one to one relationship between C source files and C modules. By default, the compiler gives this module the same name as the original C source file. Libraries of multiple C modules can only be created using XLIB.

Assembler programs can be constructed so that a single source file contains multiple modules, each of which can be a program module or a library module.

LIBRARIES

In the previous diagram, the file `library.rxx` consists of multiple modules, each of which could have been produced by the assembler or the C compiler.

The module `module_c`, which has the `PROGRAM` attribute will *always* be loaded whenever the `library.rxx` file is listed among the input files for the linker. In the run-time libraries, the startup module `cstartup` (which is a required module in all C programs) has the `PROGRAM` attribute so that it will always get included when you link a C project.

The other modules in the `library.rxx` file have the `LIBRARY` attribute. Library modules are only loaded if they contain an entry (a function, variable, or other symbol declared as `PUBLIC`) that is referenced in some way by another module that is loaded. This way, XLINK only gets the modules from the library file that it needs to build the program, and no more. For example, if the entries in `module_e` are not referenced by any loaded module, `module_e` will not be loaded.

This works as follows:

If `module_a` makes a reference to an external symbol, XLINK will search the other input files for a module containing that symbol as a `PUBLIC` entry; ie a module where the entry itself is located. If it finds the symbol declared as `PUBLIC` in `module_c`, it will then load that module (if it has not already been loaded). This procedure is iterative, so if `module_c` makes a reference to an external the same thing happens.

It is important to understand that a library file is just like any other relocatable object file. There is really no distinct type of file called a library (modules have a `LIBRARY` or `PROGRAM` attribute). What makes a file a library is what it contains and how it is used. Put simply, a library is a `.rxx` file that contains a group of related, often-used modules, most of which have a `LIBRARY` attribute so that they can be loaded on a demand-only basis.

CREATING LIBRARIES

You can create your own libraries, or add to existing libraries, using C or assembler modules. The C compiler `-b` option can be used to force a C module to have a `LIBRARY` attribute instead of the default `PROGRAM` attribute. In assembler programs, the `MODULE` directive is used to give a module the `LIBRARY` attribute, and the `NAME` directive is used to give a module the `PROGRAM` attribute.

The XLIB Librarian is used to create and manage libraries. Among other tasks, it can be used to alter the attribute (`PROGRAM`/`LIBRARY`) of any other module after it has been compiled or assembled.

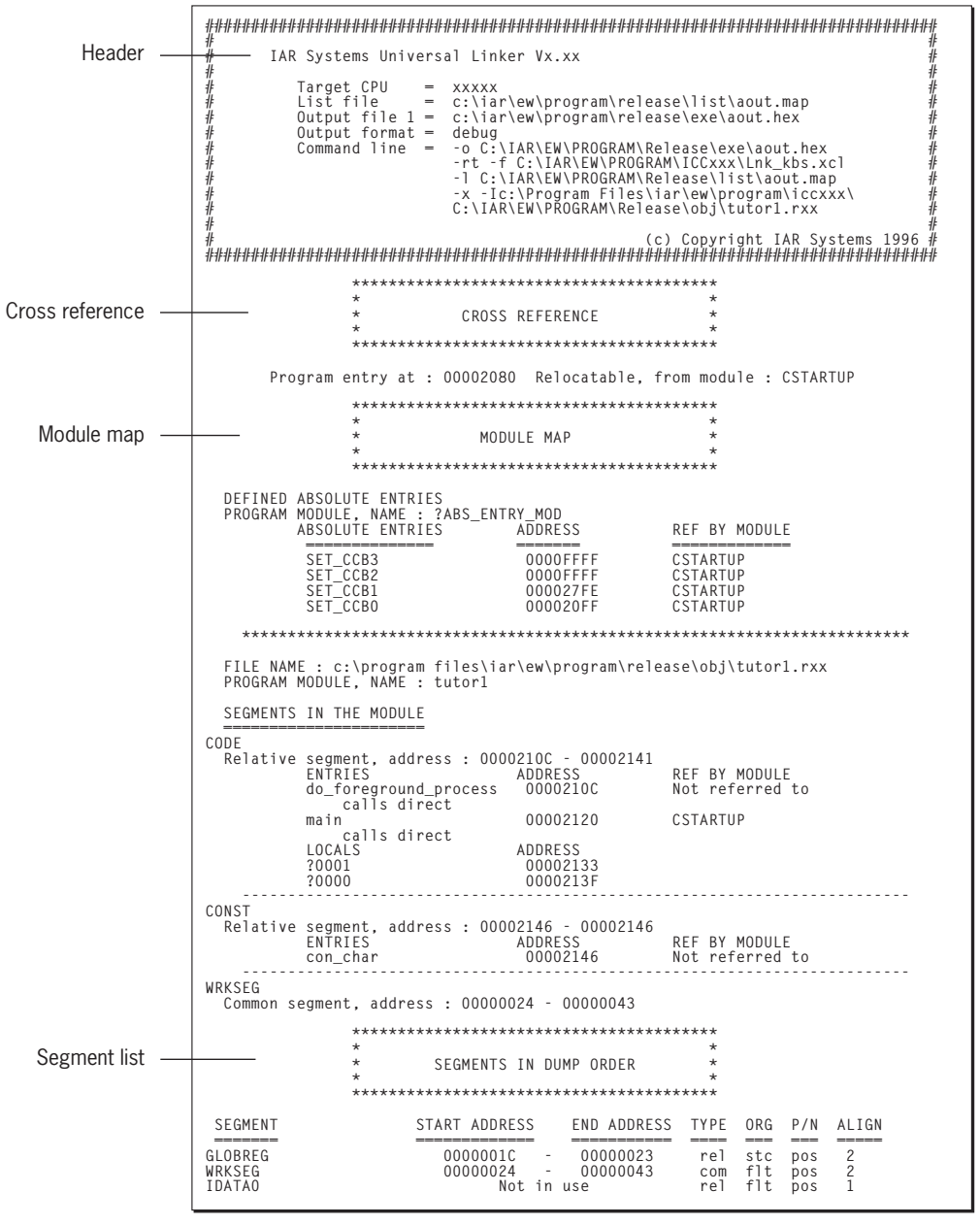
SEGMENT LOCATION

Once XLINK has identified the modules to be loaded for a program, one of its most important functions is to assign load addresses to the various code and data segments that are being used by the program.

In assembly language programs the programmer is responsible for declaring and naming relocatable segments and determining how they are used. In C programs the compiler creates and uses a set of pre-defined code and data segments, and you have only limited control over segment naming and usage.

LISTING FORMAT

The default XLINK listing format is shown below:

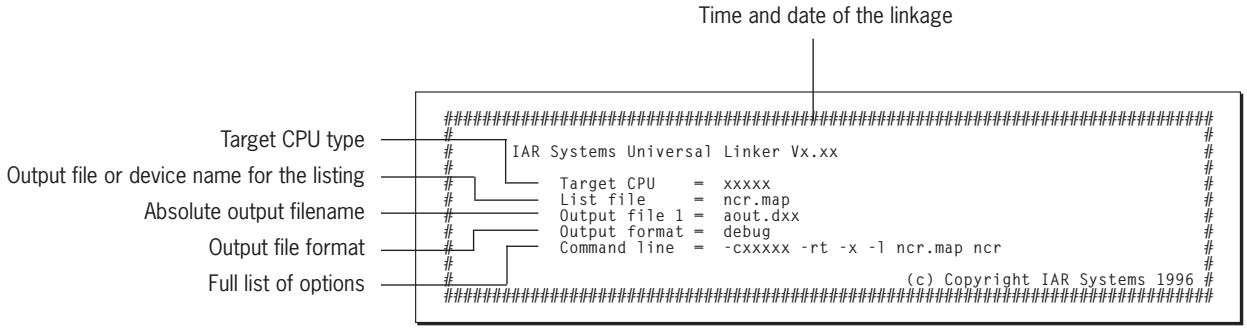



```
#############################################################################
#                                                                           #
#       IAR Systems Universal Linker Vx.xx                                  #
#                                                                           #
#          Target CPU    =  xxxxx                                           #
#          List file     =  c:\iar\ew\program\release\list\aout.map         #
#          Output file 1 =  c:\iar\ew\program\release\exe\aout.hex           #
#          Output format =  debug                                           #
#          Command line  =  -o C:\IAR\EW\PROGRAM\Release\exe\aout.hex       #
#                           -rt -f C:\IAR\EW\PROGRAM\ICCxxx\Lnk_kbs.xcl     #
#                           -l C:\IAR\EW\PROGRAM\Release\list\aout.map      #
#                           -x -Ic:\Program Files\iar\ew\program\iccxxx\    #
#                           C:\IAR\EW\PROGRAM\Release\obj\tutor1.rxx         #
#                                                                           #
#                                                (c) Copyright IAR Systems 1996 #
#############################################################################

                  ****************************************
                  *                                      *
                  *           CROSS REFERENCE            *
                  *                                      *
                  ****************************************

        Program entry at : 00002080  Relocatable, from module : CSTARTUP

                  ****************************************
                  *                                      *
                  *             MODULE MAP               *
                  *                                      *
                  ****************************************

  DEFINED ABSOLUTE ENTRIES
  PROGRAM MODULE, NAME : ?ABS_ENTRY_MOD
          ABSOLUTE ENTRIES        ADDRESS          REF BY MODULE
          ================        =======          =============
          SET_CCB3                 0000FFFF        CSTARTUP
          SET_CCB2                 0000FFFF        CSTARTUP
          SET_CCB1                 000027FE        CSTARTUP
          SET_CCB0                 000020FF        CSTARTUP

    ***************************************************************************

  FILE NAME : c:\program files\iar\ew\program\release\obj\tutor1.rxx
  PROGRAM MODULE, NAME : tutor1

  SEGMENTS IN THE MODULE
  ======================
CODE
  Relative segment, address : 0000210C - 00002141
          ENTRIES                 ADDRESS          REF BY MODULE
          do_foreground_process    0000210C        Not referred to
              calls direct
          main                     00002120        CSTARTUP
              calls direct
          LOCALS                  ADDRESS
          ?0001                    00002133
          ?0000                    0000213F
    ---------------------------------------------------------------------------
CONST
  Relative segment, address : 00002146 - 00002146
          ENTRIES                 ADDRESS          REF BY MODULE
          con_char                 00002146        Not referred to
    ---------------------------------------------------------------------------
WRKSEG
  Common segment, address : 00000024 - 00000043

                  ****************************************
                  *                                      *
                  *        SEGMENTS IN DUMP ORDER        *
                  *                                      *
                  ****************************************

 SEGMENT               START ADDRESS    END ADDRESS  TYPE  ORG  P/N  ALIGN
 =======               =============    ===========  ====  ===  ===  =====
GLOBREG                  0000001C   -   00000023     rel  stc  pos   2
WRKSEG                   00000024   -   00000043     com  flt  pos   2
IDATA0                        Not in use            rel  flt  pos   1
```

It consists of the following sections:

HEADER

Shows the command line, and options selected for the XLINK command:



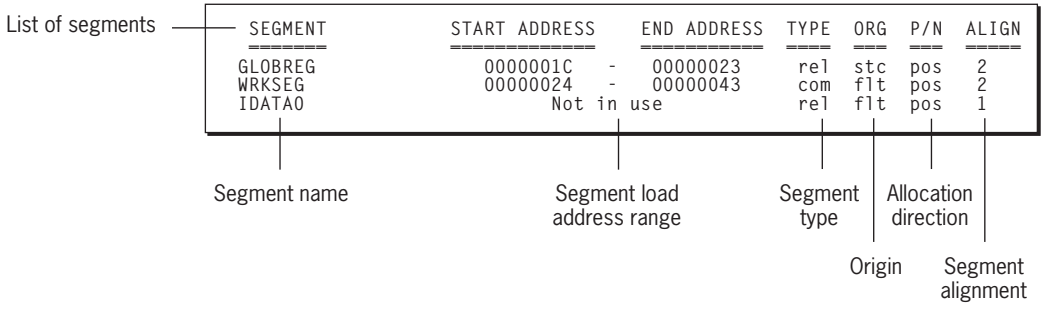
The full list of options shows the options specified on the command line. Options in command files specified with the `-f` option are also shown, in brackets.

CROSS REFERENCE

The cross reference consists of the entry list, module map and/or the segment map. It includes the program entry point, used in some output formats for hardware emulator support; see the assembler `END` directive in *Module control directives*, page 74.

Segment list (-xs)

The segment list gives the segments in the order in which they were linked:

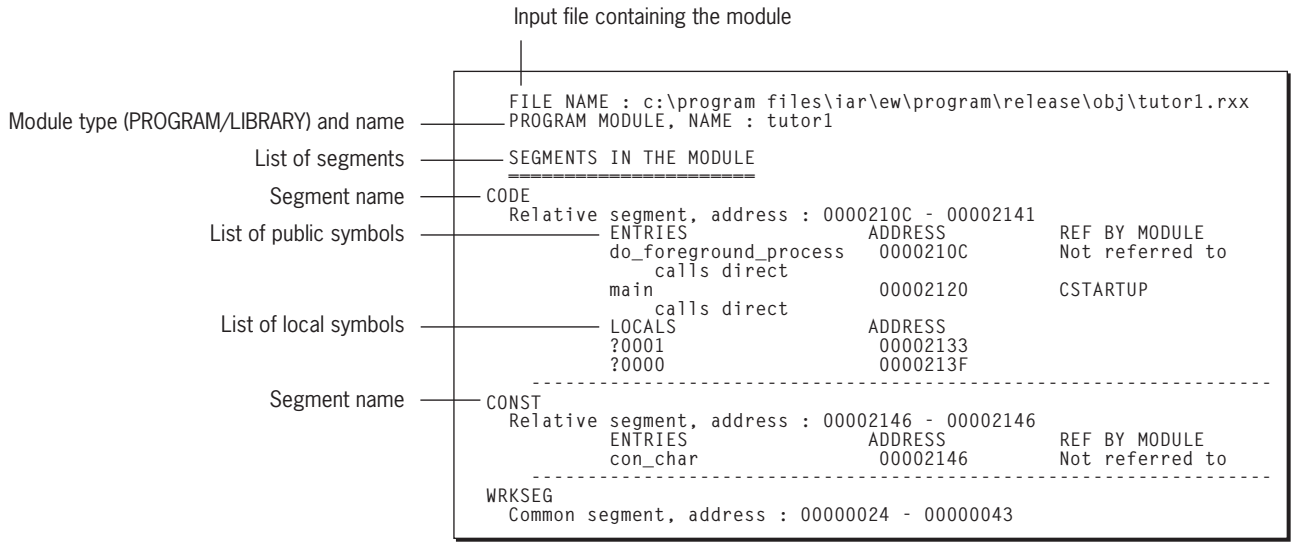


This lists the start and end address for each segment, and the following parameters:

Parameter	*Description*
TYPE	The type of segment: rel Relative stc Stack bnk Banked com Common dse Defined but not used
ORG	The origin; the type of segment start address: stc Absolute, for ASEG segments. flt Floating, for RSEG, COMMON, or STACK segments.
P/N	Positive/Negative; how the segment is allocated: pos Upwards, for ASEG, RSEG, or COMMON segments. neg Downwards, for STACK segments.
ALIGN	The segment is aligned to the next 2^ALIGN address boundary.

Module listing (-xm)

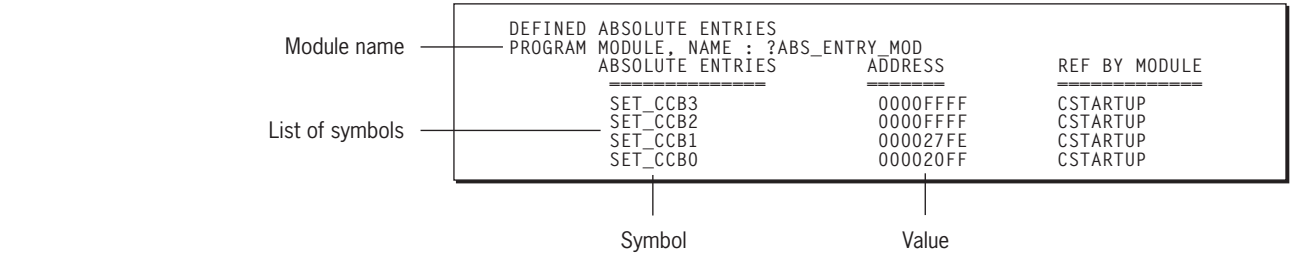
The module map consists of a subsection for each module that was loaded as part of the program. Each subsection shows the following information:



For each segment the module map also lists locals and entries.

Symbol listing (-xe)

Shows the entry name and address for each module and filename.



XLINK OPTIONS SUMMARY

XLINK options allow you to control the operation of XLINK from the command line.

The options are divided into the following sections:

Output	Include
#define	Target
Error	Miscellaneous
List	Segment control

SETTING XLINK OPTIONS

To set options from the command line, either:

- Specify the options on the command line, after the `xlink` command.
- Specify the options in an XCL command file, and include this on the command line with `-f` *`file`* command.
- Specify the options in the `XLINK_ENVPAR` environment variable; see the *Command Line Interface Guide*.

SUMMARY OF OPTIONS

The following is a summary of all the XLINK options. For a full description of any option, see under the option's category name in the next chapter, *XLINK options reference*.

Option	*Description*	*Section*
`-!`	Comment delimeter.	Miscellaneous
`-A` *`file, …`*	Load input files as program modules.	Miscellaneous
`-B`	Always generate output.	Error
`-b`*`bank_def`*	Define banked segments.	Segment control
`-C` *`file, …`*	Conditionally load input files.	Miscellaneous
`-c`*`cpu`*	Processor type.	Miscellaneous
`-D`*`symbol=value`*	Define symbol.	#define

Option	*Description*	*Section*
`-d`	Disable code generation.	Miscellaneous
`-E file, …`	Link without generating object code.	Miscellaneous
`-enew=old[,old] …`	Rename external symbols.	Miscellaneous
`-Fformat`	Output format.	Output
`-f file`	XCL filename.	Target
`-G`	No global type checking.	Error
`-Ipathname`	Include paths.	Include
`-lfile`	List.	List
`-m`	Use less host memory.	Miscellaneous
`-n`	Ignore local symbols.	Miscellaneous
`-o file`	Output filename.	Output
`-plines`	Lines/page	List
`-R`	Disable range check.	Error
`-r`	Debug info.	Output
`-rt`	Debug info with terminal I/O.	Output
`-S`	Silent operation.	Miscellaneous
`-t`	Temporary file.	Miscellaneous
`-w`	Disable warnings.	Error
`-x[ems]`	Cross reference.	List
`-Y[char]`	Format variant.	Output
`-Zseg_def`	Define segments.	Segment control
`-z`	Segment overlap warnings.	Error

XLINK OPTIONS REFERENCE

This section gives details of the XLINK options classified according to their function.

OUTPUT

The output options are used to specify the output format and the level of debugging information.

`-F`*format*	Output format.
`-o` *file*	Output filename.
`-r`	Debug info.
`-rt`	Debug info with terminal I/O.
`-Y[`*char*`]`	Format variant.

OUTPUT FORMAT (-F)

Syntax: `-F`*format*

Use **Output format** (`-F`) to select the output format.

The environment variable `XLINK_FORMAT` can be set to install an alternate default format on your system; see *XLINK_FORMAT* in the *Command Line Interface Guide*.

The parameter should be one of the supported XLINK output formats; for details of the formats see the chapter *XLINK output formats*.

If not specified, the default `INTEL-EXTENDED` format will be used.

Note that specifying the **Output format** (`-F`) option as `DEBUG` does not include C-SPY debug support. Use the **Debug info** (`-r`) option instead.

OUTPUT FILENAME (-o)

Syntax: `-o` *file*

Use **Output filename** (`-o`) to specify the name of the XLINK output file. If a name is not specified the linker will use the name `aout.hex`. If a name is supplied without a file type, the default file type for the selected output format (**Output format** (`-F`) option) will be used.

If a format is selected that generates two output files, the user-specified file type (`.axx`) will only affect the primary output file (first format).

DEBUG INFO (-r)

Syntax: `-r`

Use **Debug info** (`-r`) to output a file in DEBUG (AUBROF) format, with a `.dxx` extension, to be used with the C-SPY debugger, or emulators which support the IAR Systems DEBUG format.

Specifying **Debug info** (`-r`) overrides any **Output format** (`-F`) option.

DEBUG INFO WITH TERMINAL I/O (-rt)

Syntax: `-rt`

Use **Debug info with terminal I/O** (`-rt`) to use the output file with the C-SPY debugger and emulate terminal I/O.

FORMAT VARIANT (-Y)

Syntax: `-Y[`*`char`*`]`

Use **Format variant** (`-Y`) to select enhancements available for some output formats. For more information see the chapter *XLINK output formats*.

#define

The **#define** option allows you to define symbols.

`-D`*`symbol`*`=`*`value`* Define symbol.

DEFINE SYMBOL (-D)

Syntax: `-D`*`symbol`*`=`*`value`*

where *`symbol`* is any external (`EXTERN`) symbol in the program that is not defined elsewhere, and *`value`* the value to be assigned to *`symbol`*.

Use **Define symbol** (`-D`) to define absolute symbols at link time. This is especially useful for configuration purposes. Any number of symbols can be defined using the XCL file mode of XLINK operation. The symbol(s) defined in this manner will belong to a special module generated by the linker called `?ABS_ENTRY_MOD`.

XLINK will display an error message if you attempt to redefine an existing symbol.

ERROR

The **Error** options determine the error and warning messages generated by the XLINK Linker.

`-B`	Always generate output.
`-G`	No global type checking.
`-R`	Disable range check.
`-w`	Disable warnings.
`-z`	Segment overlap warnings.

ALWAYS GENERATE OUTPUT (-B)

Syntax: `-B`

Use **Always generate output** (`-B`) to generate an output file even if a non-fatal error was encountered during the linking process, such as a missing global entry or a duplicate declaration. Normally, XLINK will not generate an output file if an error is encountered. Note that XLINK always aborts on fatal errors, even with `-B`.

The **Always generate output** (`-B`) option allows missing entries to be patched in later in the absolute output image.

NO GLOBAL TYPE CHECKING (-G)

Syntax: `-G`

Use **No global type checking** (`-G`) to disable type checking at link time. While a well-written program should not need this option, there may be occasions where it is helpful.

By default, XLINK performs link-time type checking between modules by comparing the external references to an entry with the `PUBLIC` entry (if the information exists in the object modules involved). A warning is printed if there are mismatches; otherwise the linker will continue and not abort.

DISABLE RANGE CHECK (-R)

Syntax: `-R`

Use **Disable range check** (`-R`) to disable the address range check.

If an address is relocated out of the target CPU's address range (code, external data, or internal data address) an error message is generated. This usually indicates an error in an assembly language module or in the XLINK segment definition list (`-Z` command).

DISABLE WARNINGS (-w)

Syntax: `-w`

Use **Disable warnings** (`-w`) to suppress all warning messages. They will, however, still be counted and shown in the linker's final statistics.

SEGMENT OVERLAP WARNINGS (-z)

Syntax: `-z`

Use **Segment overlap warnings** (`-z`) to reduce segment overlap errors to warnings, making it possible to produce cross-reference maps, etc.

LIST

The **List** options determine the generation of an XLINK cross-reference listing.

`-l`*file*	List.
`-p`*lines*	Lines/page.
`-x[`*ems*`]`	Cross reference.

LIST (-l)

Syntax: `-l`*file*

Use **List** (`-l`) to generate a linker listing.

The name of the file or device to which a listing is directed. If an extension is not specified, `.lst` is used by default. However, an extension of `.map` is recommended to avoid confusing linker list files with assembler or compiler list files.

LINES/PAGE (-p)

Syntax: `-p`*`lines`*

Sets the number of lines per page for the XLINK listings to *`lines`*, which must be in the range 10 to 150.

The environment variable `XLINK_PAGE` can be set to install a default page length on your system; see *XLINK_PAGE* in the *Command Line Interface Guide*.

CROSS REFERENCE (-x)

Syntax: `-x[`*`ems`*`]`

Use **Cross reference** (`-x`) to determine the contents of the XLINK listing file. The following options are available:

Option	*Command line*	*Description*
Segment listing	`s`	A list of all the segments in dump order.
Module listing	`m`	A list of all segments, local symbols, and entries (public symbols) for every module in the program.
Symbol listing	`e`	An abbreviated list of every entry (global symbol) in every module. This entry map is useful for quickly finding the address of a routine or data element.

When this option is specified without any of the optional parameters, a default cross-reference listing will be generated which is equivalent to `-xms`. This includes:

- A header section with basic program information.
- A module load map with symbol cross-reference.
- A segment load map in dump order.

INCLUDE

The **Include** option allows you to set the include path for linker command files.

Command line

`-I`*`pathname`* Include paths.

INCLUDE PATHS (-I)

Syntax: `-I`*`pathname`*

Specifies the pathname to be searched for linker command files.

By default, XLINK searches for linker command files only in the current working directory. The **Include paths** (`-I`) option allows you to specify the names of the directories which it will also search if it fails to find the file in the current working directory

This is equivalent to the `XLINK_DFLTDIR` command line option; see the *Command Line Interface Guide*.

TARGET

The **Target** options specify the linker command file name.

Command line

`-f` *`file`* XCL filename.

XCL FILENAME (-f)

Syntax: `-f` *`file`*

Use `-f` to extend the XLINK command line by reading arguments from a command file, just as if they were typed in on the command line. If not specified an extension of `.xcl` is assumed.

Arguments are entered into the XCL file with a text editor using the same syntax as on the command line. However, in addition to spaces and tabs, the end-of-line CR is also treated as a valid delimiter between arguments. A command line may be extended by the \↵ sequence.

MISCELLANEOUS

The following additional options can be set from the command line or in XCL files:

`-! comment -!`	Comment delimiter.
`-A file, …`	Load input files as program modules.
`-C file, …`	Conditionally load input files.
`-ccpu`	Processor type.
`-d`	Disable code generation.
`-E file, …`	Link without generating object code.
`-enew=old[,old] …`	Rename external symbols.
`-m`	Use less host memory.
`-n`	Ignore local symbols.
`-S`	Silent operation.
`-t`	Temporary file.

The C compiler includes default XCL files for each chip option and memory model.

COMMENT DELIMITER (-!)

Syntax: `-! comment -!`

Use `-!` to bracket off comments in an XLINK `.xcl` file. Unless the `-!` is at the beginning of a line, it must be preceded by a space or tab.

LOAD INPUT FILES AS PROGRAM MODULES (-A)

Syntax: `-A file, …`

Use `-A` to temporarily force all of the modules within the specified input files to be loaded as if they were all program modules, even if some of the modules have the `LIBRARY` attribute.

This option is particularly suited for testing library modules before they are installed in a library file, since the `-A` option will override an existing library module with the same entries. In other words, XLINK will load the module from the *input file* specified in the `-A` argument instead of one with an entry with the same name in a library module.

For example, to load the user-written library module `putchar.r37` instead of the standard one in the `CLIB` library:

```
-! these lines are in an XCL file … -!
-A putchar
CLIB
```

This assumes that the `putchar` file contains the same global entry as one of the modules in `CLIB`.

CONDITIONALLY LOAD INPUT FILES (-C)

Syntax: `-C` *`file, …`*

Use `-C` to temporarily cause all of the modules within the specified input files to be treated as if they were all library modules, even if some of the modules have the `PROGRAM` attribute. This means that the modules in the input files will be loaded only if they contain an entry that is referenced by another loaded module.

For example, to load the user-defined `CSTARTUP` module from the file `cstartup` instead of the program module of the same name in `CLIB`:

```
-! these lines are in an XCL file -!
cstartup
-C CLIB
```

This allows you to test the `CSTARTUP` module before installing it in the library.

PROCESSOR TYPE (-c)

Syntax: `-c`*`cpu`*

Use `-c` to set the CPU type.

The environment variable `XLINK_CPU` can be set to install a default for the `-c` option so that it does not have to be specified on the command line; see *XLINK_CPU* in the *Command Line Interface Guide*.

DISABLE CODE GENERATION (-d)

Syntax: `-d`

Use `-d` to disable the generation of output code from XLINK. This option is useful for the trial linking of programs; eg checking for syntax errors, missing symbol definitions, etc. XLINK will run slightly faster for larger programs when this option is used.

LINK WITHOUT GENERATING OBJECT CODE (-E)

Syntax: `-E file, …`

Use `-E` to empty load specified input files; they will be processed normally in all regards by the linker but output code will not be generated for these files.

One potential use for this feature is in creating separate output files for programming multiple EPROMs. This is done by empty loading all input files except the ones you want to appear in the output file.

In the following example a project consists of four files, `file1` to `file4`, but we only want object code generated for `file4` to be put into an EPROM:

```
-E file1,file2,file3
file4
-o project.hex
```

RENAME EXTERNAL SYMBOLS (-e)

Syntax: `-enew=old [,old] …`

Use `-e` to configure a program at link time by redirecting a function call from one function to another.

This can also be used for creating stub functions; ie when a system is not yet complete, undefined function calls can be directed to a dummy routine until the real function has been written.

USE LESS HOST MEMORY (-m)

Syntax: `-m`

Use `-m` to reduce the amount of host system memory needed by using file pointers to all segments and modules, instead of reading all input files into RAM. If XLINK runs out of host memory during a link, this option will often help. However, XLINK will run more slowly if the `-m` option is used.

The `-m` option is equivalent to:

```
set XLINK_MEMORY=0
```

See *XLINK_MEMORY* in the *Command Line Interface Guide*.

IGNORE LOCAL SYMBOLS (-n)

Syntax: `-n`

Use `-n` to ignore all local (non-public) symbols in the input modules. This option speeds up the linking process and can also reduce the amount of host memory needed to complete a link. If `-n` is used, locals will not appear in the listing cross-reference and will not be passed on to the output file.

Note that local symbols are only included in files if they were compiled or assembled with the appropriate option to specify this.

SILENT OPERATION (-S)

Syntax: `-S`

Use `-S` to turn off the XLINK sign-on message and final statistics report so that nothing appears on your screen while it runs. However, it does not disable error and warning messages or the listing output.

TEMPORARY FILE (-t)

Syntax: `-t`

Use `-t` to force XLINK to use a temporary file, with the default name `xlink.tmp` in the current directory, to store a large part of the linker symbol tables. This can significantly reduce the amount of host system memory needed to link a program with a large number of symbols; eg more than 1500. In some cases, it may be necessary to use this option to complete a link process.

Note that the `-t` option can significantly increase the time it takes to link a program. The `-m` file-bound processing option will also be enabled automatically when `-t` is used.

The environment variable `XLINK_TFILE` can be set to an alternate filename (with drive and directory path) to use for the temporary file; see *XLINK_TFILE* in the *Command Line Interface Guide*.

SEGMENT CONTROL

These options control the allocation of segments.

`-b`*`bank_def`* Define banked segments.

`-Z`*`seg_def`* Define segments.

DEFINE BANKED SEGMENTS (-b)

Syntax: `-b [`*`addrtype`*`] [(`*`type`*`)]`
`segments=first,length,increment`

where the parameters are as follows:

`addrtype` The type of load addresses used when dumping the code:

`omitted`	Logical addresses with bank number.
`#`	Linear physical addresses.
`@`	64180-type physical addresses.

`type` Specifies the memory type for all segments in *`segments`* or *`bankedsegments`*, if applicable for the target processor. If omitted it defaults to `UNTYPED`.

`segments` The list of banked segments to be linked.

The delimiter between segments in the list determines how they are packed:

`:` (colon)	The next segment will be placed in a new bank.
`,` (comma)	The next segment will be placed in the same bank as the previous one.

`first` The start address of the first segment in the banked segment list. This is a 32-bit value: the high-order 16 bits represent the starting bank number while the low-order 16 bits represent the start address for the banks in the logical address area.

`length` The length of each bank, in bytes. This is a 16-bit value.

increment The incremental factor between banks, ie the number that will be added to *first* to get to the next bank. This is a 32-bit value: the high-order 16 bits are the bank increment, and the low-order 16 bits are the increment from the start address in the logical address area.

Use `-b` to allocate banked segments for a program that is designed for bank-switched operation. It also enables the banking mode of linker operation.

There can be more than one `-b` definition.

For example, to specify that the three code segments `BSEG1`, `BSEG2`, and `BSEG3` should be linked into banks starting at 8000, each with a length of 4000, with an increment between banks of 10000:

```
-b(CODE)BSEG1,BSEG2,BSEG3=8000,4000,10000
```

DEFINE SEGMENTS (-Z)

Syntax: `-Z [(`*type*`)] `*segments* `[=|#] [`*start-end*`,] … [`*address*`]`

where the parameters are as follows:

type Specifies the memory type for all segments in *segments* or *bankedsegments*, if applicable for the target processor. If omitted it defaults to `UNTYPED`.

segments A list of one or more segments to be linked, separated by commas.

The segments are allocated in memory in the same order as they are listed. Appending +*nnnn* to a segment name increases the amount of memory that XLINK will allocate for that segment by *nnnn* bytes.

= or #	Specifies how segments are allocated. = Allocates the segments so they begin at the start of the specified range (upwards allocation). # Allocates the segments so they finish at the end of the specified range (downwards allocation). If an allocation operator (and range) is not specified, the segments will be allocated upwards from the last segment that was linked, or from address 0 if no segments have been linked.
start, *end*	Addresses defining a range within which the listed *segments* should be placed.
address	Start address for placing any remaining segments to be allocated.

Use `-Z` to specify how and where segments will be allocated in the memory map.

If the linker finds a segment in an input file that is not defined either with `-Z` or `-b` (banked definition command), a warning will be displayed by the linker. However, the segment will still be allocated as if it were listed in the last segment definition; ie at the next available address.

There can be more than one `-Z` definition.

Additional related topics and optional forms for `-Z` are described below.

Allocation segment types
The following table lists the different types of segments that can be processed by XLINK:

Segment type	*Description*
`STACK`	Allocated from high to low addresses by default. The aligned segment size is subtracted from the load address before allocation, and successive segments are placed below the preceding segment.
`RELATIVE` `COMMON`	Allocated from low to high addresses by default.

If stack segments are mixed with relative or common segments in a segment definition, the linker will produce a warning message but will allocate the segments according to the default allocation set by the first segment in the segment list.

Common segments have a size equal to the largest declaration found for the particular segment. That is, if module `A` declares a common segment `COMSEG` with size 4, while module `B` declares this segment with size 5, the latter size will be allocated for the segment.

Be careful not to overlay common segments containing code or initializers.

Relative and stack segments have a size equal to the sum of the different (aligned) declarations.

Memory types of segments
The optional *`type`* parameter is used to assign a type to all of the segments in the list. The *`type`* parameter affects how XLINK processes the segment overlaps. Additionally, it generates information in some of the output formats that are used by some hardware emulators and by C-SPY.

Segment type	*Description*
`BIT`	Bit memory.*
`CODE`	Code memory.
`DATA`	Data memory.
`FAR`	Data in `FAR` memory. XLINK will not check access to it, and a part of a segment straddling a 64 Kbyte boundary will be moved upwards to start at the boundary.
`FARC`, `FARCONST`	Constant in `FAR` memory (behaves as above).
`FARCODE`	Code in `FAR` memory.
`HUGE`	Data in `HUGE` memory. No straddling problems.
`HUGEC`, `HUGECONST`	Constant in `HUGE` memory.
`HUGECODE`	Code in `HUGE` memory.
`NEAR`	Data in the first 64 Kbytes of memory.

Segment type	*Description*
NEARC, NEARCONST	Constant in NEAR memory.
NPAGE	Absolute-addressed data memory.
UNTYPED	Default type.
ZPAGE	Zero-page data memory.

* The address of a BIT segment is specified in bits, not in bytes. BIT memory is allocated first.

Range errors
If the ranges specified in the -Z command are too short, it will cause either error 24 Segment *segment* overlaps segment *segment*, if any segment overlaps another, or error 26 Segment *segment* is too long, if the ranges are too small.

By default, XLINK checks to be sure that the various segments that have been defined (by the -Z command and absolute segments) do not overlap in memory.

Examples
To locate SEGA at address 0, followed immediately by SEGB:

```
-Z(CODE)SEGA,SEGB=0
```

To allocate SEGA downwards from 1000H, followed by SEGB below it:

```
-Z(CODE)SEGA,SEGB#1000
```

To allocate specific areas of memory to SEGA and SEGB:

```
-Z(CODE)SEGA,SEGB=100-200,400-700,1000
```

In this example SEGA will be placed between address 100 and 200, if it fits in that amount of space. If it does not, XLINK will try the range 400–700. If none of these ranges are large enough to hold SEGA, it will start at 1000.

SEGB will be placed, according to the same rules, after segment SEGA. If SEGA fits the 100–200 range then XLINK will try to put SEGB there as well (following SEGA). Otherwise, SEGB will go into the 400 to 700 range if it is not too large, or else it will start at 1000.

XLINK OUTPUT FORMATS

This chapter gives a summary of the XLINK output formats.

SINGLE OUTPUT FILE

The following formats result in the generation of a single output file:

Format	*Type*	*Extension*	*Address type*
AOMF8051†	binary	from CPU	N
AOMFH8†	binary	from CPU	NL
AOMF8096†	binary	from CPU	N
ASHLING	binary	none	N
ASHLING-6301†	binary	from CPU	N
ASHLING-64180†	binary	from CPU	NS
ASHLING-6801†	binary	from CPU	N
ASHLING-8080†	binary	from CPU	NS
ASHLING-8085†	binary	from CPU	NS
ASHLING-Z80†	binary	from CPU	NS
DEBUG†	binary	.dbg	NL
EXTENDED-TEKHEX†	ASCII	from CPU	NLPS
HP-CODE	binary	.x	NLPS
HP-SYMB	binary	.l	NLPS
INTEL-STANDARD	ASCII	from CPU	N
INTEL-EXTENDED	ASCII	from CPU	NLPS
MILLENIUM (Tektronix)	ASCII	from CPU	N
MOTOROLA	ASCII	from CPU	NLPS
MPDS-CODE	binary	.tsk	N
MPDS-SYMB	binary	.sym	NLPS
MSD	ASCII	.sym	N

Format	*Type*	*Extension*	*Address type*
NEC-SYMBOLIC†	ASCII	.sym	N
NEC2-SYMBOLIC†	ASCII	.sym	N
NEC78K-SYMBOLIC†	ASCII	.sym	N
PENTICA-A	ASCII	.sym	NLPS
PENTICA-B	ASCII	.sym	NLPS
PENTICA-C	ASCII	.sym	NLPS
PENTICA-D	ASCII	.sym	NLPS
RCA	ASCII	from CPU	N
SYMBOLIC	ASCII	from CPU	NLPS
SYSROF†	binary	.abs	NLPS
TEKTRONIX (Millenium)	ASCII	.hex	N
TI7000 (TMS7000)	ASCII	from CPU	N
TYPED	ASCII	from CPU	NLPS
ZAX	ASCII	from CPU	NLPS

† format depends on the typing of the segments; ie the *type* field specified in the XLINK -Z option is important.

Address type
The address type is one of the following:

N = Non-banked address.

L = Banked logical address.

P = Banked physical address.

S = Banked 64180 physical address.

TWO OUTPUT FILES

The following formats result in the generation of two output files:

Format	*Code format*	*Exten.*	*Symbolic format*	*Exten.*
DEBUG-MOTOROLA	DEBUG	.a*xx*	MOTOROLA	.obj
DEBUG-INTEL-STD	DEBUG	.a*xx*	INTEL-STD	.hex
DEBUG-INTEL-EXT	DEBUG	.a*xx*	INTEL-EXT	.hex
HP	HP-CODE	.x	HP-SYMB	.l
MPDS	MPDS-CODE	.tsk	MPDS-SYMB	.sym
MPDS-I	INTEL-STANDARD	.hex	MPDS-SYMB	.sym
MPDS-M	MOTOROLA	.s19	MPDS-SYMB	.sym
MSD-I	INTEL-STANDARD	.hex	MSD	.sym
MSD-M	MOTOROLA	.hex	MSD	.sym
MSD-T	MILLENIUM	.hex	MSD	.sym
NEC	INTEL-STANDARD	.hex	NEC-SYMB	.sym
NEC2	INTEL-STANDARD	.hex	NEC2-SYMB	.sym
PENTICA-AI	INTEL-STANDARD	.obj	PENTICA-A	.sym
PENTICA-AM	MOTOROLA	.obj	PENTICA-A	.sym
PENTICA-BI	INTEL-STANDARD	.obj	PENTICA-B	.sym
PENTICA-BM	MOTOROLA	.obj	PENTICA-B	.sym
PENTICA-CI	INTEL-STANDARD	.obj	PENTICA-C	.sym
PENTICA-CM	MOTOROLA	.obj	PENTICA-C	.sym
PENTICA-DI	INTEL-STANDARD	.obj	PENTICA-D	.sym
PENTICA-DM	MOTOROLA	.obj	PENTICA-D	.sym
ZAX-I	INTEL-STANDARD	.hex	ZAX	.sym
ZAX-M	MOTOROLA	.hex	ZAX	.sym

OUTPUT FORMAT VARIANTS

The following enhancements can be selected for the specified output formats, using the **Format variant** (`-Y`) option:

Output format	*Option*	*Description*
`PENTICA-A,B,C,D` and `MPDS-SYMB`	`Y0`	Symbols as `modules:symbolname`.
	`Y1`	Labels and lines as *`module:symbolname`*.
	`Y2`	Lines as *`module:symbolname`*.
`AOMF8051`	`Y0`	Extra type of information for Hitex.
`INTEL-STANDARD`	`Y0`	End only with `:00000001FF`.
	`Y1`	End with `PGMENTRY`, else `:0000001FF`.
`MPDS-CODE`	`Y0`	Fill with `0xFF` instead.
`DEBUG`, `-r`	`Y#`	Old UBROF version.
`INTEL-EXTENDED`	`Y0`	Segmented variant.
	`Y1`	32-bit linear variant.

Refer to the file `XLINK.DOC` for additional options that are available.

XLIB LIBRARIAN

This chapter describes the XLIB Librarian, which is designed to allow you to create and maintain relocatable libraries of routines.

INTRODUCTION

Like the XLINK Linker, the XLIB Librarian uses the UBROF standard object format (Universal Binary Relocatable Object Format) to allow it to support a wide range of 32-bit byte-oriented processors (applies to almost all current major microprocessors).

LIBRARIES

A library is a single file that contains a number of relocatable object modules, each of which can be loaded independently from other modules in the file as it is needed.

Normally, the modules in a library file all have the `LIBRARY` attribute, which means that they will only be loaded by the linker if they are actually needed in the program. This is referred to as demand loading of modules.

On the other hand, a module with the `PROGRAM` attribute is *always* loaded when the file in which it is contained is processed by the linker.

A library file is no different from any other relocatable object file produced by the assembler or C compiler, except that it includes a number of modules of the `LIBRARY` type.

USING LIBRARIES WITH C PROGRAMS

All C programs make use of libraries, and the IAR Systems C Compilers are supplied with a number of standard library files.

Most C programmers will use the XLIB Librarian at some point, for one of the following reasons:

- To replace or modify a module in one of the standard libraries. For example, the librarian can be used to replace the distribution versions of the `CSTARTUP` and/or `putchar` modules with ones that you have customized.

- To add C or assembler modules to the standard library file so they will always be available whenever a C program is linked.
- To create custom library files that can be linked into their programs, as needed, along with the standard C library.

USING LIBRARIES WITH ASSEMBLER PROGRAMS

If you are only using assembler there is no need to use libraries. However, libraries provide the following advantages, especially when writing medium- and large-sized assembler applications:

- They allow you to combine utility modules used in more than one project into a simple library file. This simplifies the linking process by eliminating the need to include a list of input files for all the modules you need. Only the library module(s) needed for the program will be included in the output file.
- They simplify program maintenance by allowing multiple modules to be placed in a single assembler source file. Each of the modules can be loaded independently as a library module.
- They reduce the number of object files that make up an application, maintenance, and documentation.

You can create your assembly language library files using one of two basic methods:

- A library file can be created by assembling a single assembler source file which contains multiple library-type modules. The resulting library file can then be modified using XLIB.
- A library file can be produced by using the XLIB Librarian to merge any number of existing modules together to form a user-created library.

The `NAME` and `MODULE` assembler directives are used to declare modules as being of `PROGRAM` or `LIBRARY` type, respectively.

XLIB COMMAND SUMMARY

This chapter summarizes the librarian commands, classified according to their function.

A full alphabetical reference list of commands is given in the next chapter.

LIBRARY LISTING COMMANDS

LIST-ALL-SYMBOLS	Lists every symbol in modules.
LIST-CRC	Lists CRC values of modules.
LIST-DATE-STAMPS	Lists dates of modules.
LIST-ENTRIES	Lists PUBLIC symbols in modules.
LIST-EXTERNALS	Lists EXTERN symbols in modules.
LIST-MODULES	Lists modules.
LIST-OBJECT-CODE	Lists low-level relocatable code.
LIST-SEGMENTS	Lists segments in modules.

LIBRARY EDITING COMMANDS

DELETE-MODULES	Removes modules from a library.
FETCH-MODULES	Adds modules to a library.
INSERT-MODULES	Moves modules in a library.
MAKE-LIBRARY	Changes a module to library type.
MAKE-PROGRAM	Changes a module to program type.
RENAME-ENTRY	Renames PUBLIC symbols.
RENAME-EXTERNAL	Renames EXTERN symbols.
RENAME-GLOBAL	Renames EXTERN and PUBLIC symbols.
RENAME-MODULE	Renames one or more modules.

RENAME-SEGMENT	Renames one or more segments.
REPLACE-MODULES	Updates executable code.

MISCELLANEOUS LIBRARY COMMANDS

COMPACT-FILE	Shrinks library file size.
DEFINE-CPU	Specifies CPU type.
DIRECTORY	Displays available object files.
DISPLAY-OPTIONS	Displays XLIB options.
ECHO-INPUT	Command file diagnostic tool.
EXIT	Returns to operating system.
HELP	Displays help information.
ON-ERROR-EXIT	Quits on a batch error.
QUIT	Returns to operating system.
REMARK	Comment in command file.

XLIB COMMAND REFERENCE

This chapter gives a full syntactic and functional description of all librarian commands.

The individual words of an identifier can be abbreviated to the limit of ambiguity. For example, `LIST-MODULES` can be abbreviated to `L-M`.

When running XLIB you can press ↵ at any time to prompt for information, or display a list of the possible options.

PARAMETERS

The following parameters are common to many of the XLIB commands.

Parameter	*What it means*	
objectfile	File containing object modules.	
start, end	The first and last modules to be processed, in one of the following forms:	
	n	The *n*th module.
	$	The last module.
	name	Module *name*.
	name+*n*	The module *n* modules after *name*.
	$-*n*	The module *n* modules before the last.
listfile	File to which a listing will be sent.	
source	A file from which modules will be read.	
destination	The file to which modules will be sent.	

MODULE EXPRESSIONS

In most of the XLIB commands you can or must specify a source module (like *oldname* in `RENAME-MODULE`), or a range of modules (*startmodule*, *endmodule*).

Internally in all XLIB operations modules are numbered upwards from one. Modules may be referred to by the actual name of the module, by the name plus or minus a relative expression, or by an absolute number. The latter is very useful when a module name is very long, unknown, or contains unusual characters (like space or comma). Below is a list of the available variations on module expressions:

Name	*Description*
`3`	The third module.
`$`	The last module.
`name``+4`	The module 4 modules after *`name`*.
`name``-12`	The module 12 modules before *`name`*.
`$-2`	The module 2 modules before the last module.

The command `LIST-MOD FILE,,$-2` will thus list the three last modules in `FILE` on the terminal.

LIST FORMAT

The `LIST` commands give a list of symbols, where each symbol has one of the following prefixes:

Prefix	*Description*
`nn``.Pgm`	A program module with relative number *`nn`*.
`nn``.Lib`	A library module with relative number *`nn`*.
`Ext`	An external in the current module.
`Ent`	An entry in the current module.
`Loc`	A local in the current module.
`Rel`	A standard segment in the current module.
`Stk`	A stack segment in the current module.
`Com`	A common segment in the current module.

COMPACT-FILE

Shrinks library file size.

SYNTAX

`COMPACT-FILE` *`objectfile`*

DESCRIPTION

Use `COMPACT-FILE` to concatenate short, absolute records into longer records of variable length. This will decrease the size of a library file by about 5 %, to give library files which take up less time during the loader/linker process.

EXAMPLES

The following command compacts the file `maxmin.rxx`:

```
COMPACT-FILE maxmin ↵
```

This displays:

```
20 byte(s) deleted
```

DEFINE-CPU

Specifies CPU type.

SYNTAX

`DEFINE-CPU` *`cpu`*

PARAMETERS

`cpu` The target processor.

DESCRIPTION

This command must be issued before any operations on object files can be done.

EXAMPLES

The following command defines the CPU as *`cpu`*:

```
DEF-CPU cpu ↵
```

The environment variable XLIB_CPU can be set to define a default value for the DEFINE-CPU command so that it does not have to be specified within XLIB; see XLIB_CPU in *H8 Command Line Interface Guide*.

DELETE-MODULES

Removes modules from a library.

SYNTAX

DELETE-MODULES *objectfile start end*

DESCRIPTION

Use DELETE-MODULES to delete the specified modules.

EXAMPLES

The following command deletes module 2 from the file math.r*xx*:

```
DEL-MOD math 2 2 ↵
```

DIRECTORY

Displays available object files.

SYNTAX

DIRECTORY [*specifier*]

DESCRIPTION

Use DIRECTORY to display on the terminal all files of the type that applies to the target processor. If no *specifier* is given, the current directory is listed.

EXAMPLES

The following command lists object files in the current directory:

```
DIR ↵
```

It displays:

```
general      770
math         502
maxmin       375
```

DISPLAY-OPTIONS

Displays XLIB options.

SYNTAX

`DISPLAY-OPTIONS [`*`listfile`*`]`

DESCRIPTION

Use `DISPLAY-OPTIONS` to list on the *`listfile`* the names of all the CPUs which are recognized by this version of XLIB. The default file types of object files for the different CPUs are also listed. After that a list of all UBROF tags is output.

EXAMPLES

To list the options to the file `opts.lst`:

```
DISPLAY-OPTIONS opts ↵
```

ECHO-INPUT

Command file diagnostic tool.

SYNTAX

`ECHO-INPUT`

DESCRIPTION

`ECHO-INPUT` is useful when debugging command files in batch mode because it makes all command input visible on the terminal. In the interactive mode it has no effect.

EXAMPLES

In a batch file

```
ECHO-INPUT
```

echoes all subsequent XLIB commands.

EXIT

Returns to operating system.

SYNTAX

```
EXIT
```

DESCRIPTION

Use `EXIT` to exit from XLIB after an interactive session.

EXAMPLES

To exit from XLIB:

```
EXIT ↵
```

FETCH-MODULES

Adds modules to a library.

SYNTAX

`FETCH-MODULES` *`source destination`* [*`start`*] [*`end`*]

DESCRIPTION

Use `FETCH-MODULES` to append the specified modules to the *`destination`* file. If *`destination`* already exists, it must be empty or contain valid object modules; otherwise it will be created.

EXAMPLES

The following command copies module `mean` from `math.r`*`xx`* to `general.r`*`xx`*:

```
FETCH-MOD math general mean ↵
```

HELP

Displays help information

SYNTAX

`HELP` [*`command`*] [*`listfile`*]

PARAMETERS

command Command for which help is displayed.

DESCRIPTION

If the HELP command is given with no parameters, a list of the available commands will be displayed on the terminal. If a parameter is specified, all commands which match the parameter will be displayed with a brief explanation of their syntax and function. A * matches all commands. HELP output can be directed to any file.

EXAMPLES

For example, the command:

```
HELP LIST-MOD ↵
```

displays:

```
LIST-MODULES <Object file> [<List file>] [<Start module>]
[<End module>]
    List the module names from [<Start module>] to
    [<End module>].
```

INSERT-MODULES

Moves modules in a library.

SYNTAX

```
INSERT-MODULES objectfile start end {BEFORE | AFTER} dest
```

DESCRIPTION

Use INSERT-MODULES to move the specified modules before or after the *dest*.

EXAMPLES

The following command moves the module mean before module min in the file math.r*xx*:

```
INSERT-MOD math mean mean BEFORE min ↵
```

LIST-ALL-SYMBOLS

Lists every symbol in modules.

SYNTAX

`LIST-ALL-SYMBOLS` *`objectfile`* [*`listfile`*] [*`start`*] [*`end`*]

DESCRIPTION

Use `LIST-ALL-SYMBOLS` to list all symbols (module names, segments, externals, entries, and locals) for the specified modules in the *`objectfile`*. They are listed to the *`listfile`*.

Each symbol is identified with a prefix; see *List format*, page 194.

EXAMPLES

The following command lists all the symbols in `math.r`*`xx`*:

```
LIST-ALL-SYMBOLS math ↵
```

This displays:

```
1.  Lib  max
      Rel   CODE
      Ent   max
      Loc   A
      Loc   B
      Loc   C
      Loc   ncarry
2.  Lib  mean
      Rel   DATA
      Rel   CODE
      Ext   max
      Loc   A
      Loc   B
      Loc   C
      Loc   main
      Loc   start
3.  Lib  min
      Rel   CODE
      Ent   min
      Loc   carry
```

LIST-CRC

Lists CRC values of modules.

SYNTAX

`LIST-CRC objectfile [listfile] [start] [end]`

DESCRIPTION

Use `LIST-CRC` to list the module names and their associated CRCs for the specified modules.

Each symbol is identified with a prefix; see *List format*, page 194.

EXAMPLES

The following command lists the CRCs for all modules in `math.rxx`:

```
LIST-CRC math ↵
```

This displays:

```
EC41            1.  Lib  max
ED72            2.  Lib  mean
9A73            3.  Lib  min
```

LIST-DATE-STAMPS

Lists dates of modules.

SYNTAX

`LIST-DATE-STAMPS objectfile [listfile] [start] [end]`

DESCRIPTION

Use `LIST-DATE-STAMPS` to list the module names and their associated generation dates for the specified modules.

Each symbol is identified with a prefix; see *List format*, page 194.

EXAMPLES

The following command lists the date stamps for all the modules in `math.rxx`:

```
LIST-DATE-STAMPS math ↵
```

This displays:

```
09/Jan/96       1.  Lib  max
09/Jan/96       2.  Lib  mean
09/Jan/96       3.  Lib  min
```

LIST-ENTRIES

Lists PUBLIC symbols in modules.

SYNTAX

LIST-ENTRIES *objectfile* [*listfile*] [*start*] [*end*]

DESCRIPTION

Use LIST-ENTRIES to list the names and associated entries for the specified modules.

Each symbol is identified with a prefix; see *List format*, page 194.

EXAMPLES

The following command lists the entries for all the modules in math.r*xx*:

```
LIST-ENTRIES math ↵
```

This displays:

```
1.  Lib  max
      Ent   max
2.  Lib  mean
3.  Lib  min
      Ent   min
```

LIST-EXTERNALS

Lists EXTERN symbols in modules.

SYNTAX

LIST-EXTERNALS *objectfile* [*listfile*] [*start*] [*end*]

DESCRIPTION

Use LIST-EXTERNALS to list the module names and associated externals for the specified modules.

Each symbol is identified with a prefix; see *List format*, page 194.

EXAMPLES

The following command lists the externals for all the modules in math.r*xx*:

```
LIST-EXT math ↵
```

This displays:

```
1.  Lib  max
2.  Lib  mean
      Ext   max
3.  Lib  min
```

LIST-MODULES

Lists modules.

SYNTAX

LIST-MODULES *objectfile* [*listfile*] [*start*] [*end*]

DESCRIPTION

Use LIST-MODULES to list the module names for the specified modules.

Each symbol is identified with a prefix; see *List format*, page 194.

EXAMPLES

The following command lists all the modules in math.r*xx*:

```
LIST-MOD math ↵
```

It produces the following output:

```
1.  Lib  max
2.  Lib  min
3.  Lib  mean
```

LIST-OBJECT-CODE

Lists low-level relocatable code.

SYNTAX

`LIST-OBJECT-CODE` *`objectfile`* [*`listfile`*]

DESCRIPTION

Use `LIST-OBJECT-CODE` to list the contents of the *`objectfile`* on the *`listfile`* in an ASCII format.

Each symbol is identified with a prefix; see *List format*, page 194.

EXAMPLES

The following command lists the object code of `math.r`*`xx`* to `object.lst`:

```
LIST-OBJECT-CODE math object ↵
```

LIST-SEGMENTS

Lists segments in modules.

SYNTAX

`LIST-SEGMENTS` *`objectfile`* [*`listfile`*] [*`start`*] [*`end`*]

DESCRIPTION

Use `LIST-SEGMENTS` to list the module names and associated segments for the specified modules.

Each symbol is identified with a prefix; see *List format*, page 194.

EXAMPLES

The following command lists the segments in module `mean` in the file `math.r`*`xx`*:

```
LIST-SEG math,,mean mean ↵
```

Note the use of two commas to skip the *`listfile`* parameter.

This produces the following output:

```
2.  Lib  mean
       Rel   DATA
       Rel   CODE
```

MAKE-LIBRARY

Changes a module to library type.

SYNTAX

`MAKE-LIBRARY` *`objectfile`* [*`start`*] [*`end`*]

DESCRIPTION

Use `MAKE-LIBRARY` to change the module header attributes to conditionally loaded for the specified modules.

EXAMPLES

The following command converts all the modules in `main.r`*`xx`* to library modules:

```
MAKE-LIB main ↵
```

MAKE-PROGRAM

Changes a module to program type.

SYNTAX

`MAKE-PROGRAM` *`objectfile`* [*`start`*] [*`end`*]

DESCRIPTION

Use `MAKE-PROGRAM` to change the module header attributes to unconditionally loaded for the specified modules.

EXAMPLES

The following command converts module `start` in `main.r`*`xx`* into a program module:

```
MAKE-PROG main start ↵
```

ON-ERROR-EXIT

Quits on a batch error.

SYNTAX

```
ON-ERROR-EXIT
```

DESCRIPTION

Use `ON-ERROR-EXIT` to make the librarian abort if an error is found. Most suited for use in batch mode.

EXAMPLES

The following batch file aborts if the `FETCH-MODULES` command fails:

```
ON-ERROR-EXIT
FETCH-MODULES math new
```

QUIT

Returns to operating system.

SYNTAX

```
QUIT
```

DESCRIPTION

Use `QUIT` to exit and return to the operating system.

EXAMPLES

To quit from XLIB:

```
QUIT ↵
```

REMARK

Comment in command file.

SYNTAX

```
REMARK text
```

DESCRIPTION

Use REMARK to include a comment.

EXAMPLES

The following example illustrates the use of a comment in an XLIB command file:

```
REM Now compact file
COMPACT-FILE math
```

RENAME-ENTRY

Renames PUBLIC symbols.

SYNTAX

RENAME-ENTRY *objectfile old new* [*start*] [*end*]

DESCRIPTION

Use RENAME-ENTRY to rename all occurrences of an entry from *old* to *new* in the specified modules.

EXAMPLES

The following command renames the entry for modules 2 to 4 in math.r*xx* from mean to average:

```
RENAME-ENTRY math mean average 2 4 ↵
```

RENAME-EXTERNAL

Renames EXTERN symbols.

SYNTAX

RENAME-EXTERNAL *objectfile old new* [*start*] [*end*]

DESCRIPTION

Use RENAME-EXTERNAL to rename all occurrences of an external from *old* to *new* in the specified modules.

EXAMPLES

The following command renames all external symbols in `math.rxx` from `error` to `err`:

```
RENAME-EXT math error err ↵
```

RENAME-GLOBAL

Renames `EXTERN` and `PUBLIC` symbols.

SYNTAX

`RENAME-GLOBAL` *`objectfile old new`* [*`start`*] [*`end`*]

DESCRIPTION

Use `RENAME-GLOBAL` to rename all occurrences of an external or entry from *`old`* to *`new`* in the specified modules.

EXAMPLES

The following command renames all occurrences of `mean` to `average` in `math.rxx`:

```
RENAME-GLOBAL math mean average ↵
```

RENAME-MODULE

Renames one or more modules.

SYNTAX

`RENAME-MODULE` *`objectfile old new`*

DESCRIPTION

Use `RENAME-MODULE` to rename a module. Note that if there is more than one module with name *`old`*, only the first encountered is changed.

EXAMPLES

The following example renames the module `average` to `mean` in the file `math.rxx`:

```
RENAME-MOD math average mean ↵
```

RENAME-SEGMENT

Renames one or more segments.

SYNTAX

`RENAME-SEGMENT objectfile old new [start] [end]`

DESCRIPTION

Use `RENAME-SEGMENT` to rename all occurrences of a segment from name *`old`* to *`new`* in the specified modules.

EXAMPLES

The following example renames all `CODE` segments to `ROM` in the file `math.rxx`:

```
RENAME-SEG math CODE ROM ↵
```

REPLACE-MODULES

Updates executable code.

SYNTAX

`REPLACE-MODULES source destination`

DESCRIPTION

Use `REPLACE-MODULES` to replace modules with the same name from *`source`* to *`destination`*. All replacements are logged on the terminal. The main application for this command is to update large run-time libraries etc.

EXAMPLES

The following example replaces modules in `math.rxx` with modules from `newmath.rxx`:

```
REPLACE-MOD math newmath ↵
```

This displays:

```
Replacing module 'max'
Replacing module 'mean'
Replacing module 'min'
```

ASSEMBLER DIAGNOSTICS

This chapter lists the errors and warnings for the H8 Assembler. For details of the XLINK Linker and XLIB Librarian error messages see the chapters *XLINK diagnostics*, and *XLIB diagnostics*.

INTRODUCTION

Error messages are printed on the terminal, as well as on the optional list file.

All errors are issued as complete, self-explanatory messages. For example:

```
        ADS    B,C
-----------^
"testfile.sxx",4  Error[40]: bad instruction
```

The error message consists of the erroneous source line, with a pointer to the faulty spot, followed by the diagnostic and source line number. If include files are used, error messages will be preceded by the source line number and name of *current* file:

```
        ADS    B,C
-----------^
"subfile.h",4  Error[40]: bad instruction
```

The error messages produced by the assembler fall into six categories:

- Command line error messages.
- Assembly warning messages.
- Assembly error messages.
- Assembly fatal error messages.
- Memory overflow messages.
- Assembler internal error messages.

COMMAND LINE ERROR MESSAGES

Command line errors occur when the assembler is invoked with bad parameters. The most common situation is when a file cannot be opened, or with duplicate, mis-spelled, or missing command line switches. The messages are self-explanatory.

ASSEMBLY WARNING MESSAGES

Assembly warning messages are produced when the assembler has found a construct which probably is due to a programming error or omission. These are listed in the section *Warning messages*, page 213.

ASSEMBLY ERROR MESSAGES

Assembly error messages are produced when the assembler has found a construct which violates the language rules. These are listed in the section *Error messages*, page 215.

ASSEMBLY FATAL ERROR MESSAGES

Assembly fatal error messages are produced when the assembler has found a user error so severe that further processing is not considered meaningful. After the diagnostic message has been issued the assembly is immediately terminated. The fatal error messages are identified as 'Fatal' in the error messages list.

MEMORY OVERFLOW MESSAGES

The assembler is a memory-based program that in the case of a system with a small primary memory or in the case of very large source files may run out of memory. This is identified by the special message:

```
* * * ASSEMBLER OUT OF MEMORY * * *

Dynamic memory used: nnnnnn bytes
```

If such a situation occurs the solution is either to add system memory or to split source files into smaller modules. However, with 1 Mbyte RAM the assembler capacity should be sufficient for all reasonably sized source files.

ASSEMBLER INTERNAL ERROR MESSAGES

During assembly a number of internal consistency checks are performed and if any of these checks fail the assembler will terminate after giving a short description of the problem. Such errors should normally not occur and should be reported to the IAR Systems technical support group. Please include all possible information about the problem and, preferably, a disk containing a copy of the program that generated the internal error.

WARNING MESSAGES

GENERAL

The following table lists the general warning messages:

No	*Warning message*	*Suggestion*
0	`Unreferenced label`	The label was not used as an operand nor was it declared public.
1	`Nested comment`	A C comment was nested.
2	`Unknown escape sequence`	A backslash (\) found in a character constant or string literal was followed by an unknown escape character.
3	`Non-printable character`	A non-printable character was found in a literal or character constant.
4	`Macro or define expected`	
5	`Floating point value out-of-range`	Floating point value is too large to be represented by the floating point system of the target.
6	`Floating point division by zero`	
7	`Wrong usage of string operator ('#' or '##'); ignored.`	The current implementation restricts use of the # and ## operators to the token field of parameterized macros. In addition, the # operator must precede a formal parameter.

No	*Warning message*	*Suggestion*
8	`Macro parameter(s) not used`	
9	`Macro redefined`	
10	`Unknown macro`	
11	`Empty macro argument`	
12	`Recursive macro`	
13	`Redefinition of Special Function Register`	The `SFR` has already been defined.
14	`Division by zero`	Division by 0 in constant expression.
15	`Constant truncated`	The constant was longer than the size of the destination.
16	`Suspicious sfr expression`	A Special Function Register `SFR` is used in an expression, and the assembler cannot check access rights.
17	`Empty module '<module name>', module skipped`	An empty module was created by using `END` directly after `ENDMOD` or `MODULE`, followed by `ENDMOD` with no statements in between.
18	`End of program while in include file`	The program ended while a file was being included.
19	`Symbol '<symb>' duplicated`	
20	`Bit symbol cannot be used as operand`	A symbol was declared using the `bit` directive, but since the bit address is not calculated the symbol should not be used.

H8 ASSEMBLER

In addition to the general warnings the H8 Assembler can generate the following warnings:

No	*Error message*	*Suggestion*
400	`Number too large`	The number does not fit the instruction.
401	`Displacement out of bounds`	Branch outside `-32766` to `32768`.
402	`Address out of bounds`	The address size does not fit the instruction.
403	`Instruction may affect only part of SFR`	The SFR is wider than the instruction indicates.
404	`Code at odd address`	Self explanatory.
405	`Size of SFR undefined, WORD is assumed`	Self explanatory.
406	`Jump to odd address`	Self explanatory.
407	`Jump to address that is not multiple of 4`	Self explanatory.

ERROR MESSAGES

GENERAL

The following table lists the general error messages:

No	*Error message*	*Suggestion*
0	`Invalid syntax`	The assembler could not decode the expression.
1	`Too deep #include nesting (max. is 10)`	Fatal. Assembler limit for nesting of `#include` files exceeded. Recursive `#include` could be the reason.

No	*Error message*	*Suggestion*
2	`Failed to open #include file 'name'`	Fatal. Could not open a `#include` file. File does not exist in specified directories. Check `-I` prefixes.
3	`Invalid #include file name`	Fatal. `#include` file name must be written `<file>` or `"file"`.
4	`Unexpected end of file encountered`	Fatal. End of file encountered within a conditional assembly, the repeat directive or during macro expansion. Probable cause is a missing end of conditional assembly etc.
5	`Too long source line (max. is 512 characters) truncated`	Source line length exceeds assembler limit.
6	`Bad constant`	Character that is not a legal digit was encountered.
7	`Hexadecimal constant without digits`	Prefix `0x` or `0X` of hexadecimal constant found without following hexadecimal digits.
8	`Invalid floating point constant`	Too large or invalid syntax of floating-point constant.
9	`Too many errors encountered (>100).`	
10	`Space or tab expected`	
11	`Too deep block nesting (max is 50)`	Preprocessor directives are nested too deep.
12	`String too long (max is 509)`	Assembler string length limit exceeded.
13	`Missing delimiter in literal or character constant`	No closing delimiter `'` or `"` was found in character or literal constant.

No	*Error message*	*Suggestion*
14	`Missing #endif`	A `#if`, `#ifdef`, or `#ifndef` was found but had no matching `#endif`.
15	`Invalid character encountered: <char>; ignored`	
16	`Identifier expected`	A name of a label or symbol was expected.
17	`')' expected`	
18	`No such pre-processor command: <command>`	# was followed by an unknown identifier.
19	`Unexpected token found in pre-processor line`	The pre-processor line was not empty after the argument part was read.
20	`Argument to #define too long (max is <max>)`	
21	`Too many formal parameters for #define (max is 127)`	
22	`Macro parameter <parameter> redefined`	A `#define` symbol's formal parameter was repeated.
23	`',' or ')' expected`	
24	`Unmatched #else, #endif or #elif`	Fatal. Missing `#if`, `#ifdef`, or `#ifndef`.
25	`#error <error>.`	Printout via the `#error` directive.
26	`'(' expected`	
27	`Too many active macro parameters (max is 256)`	Fatal. Pre-processor limit exceeded.
28	`Too many nested parameterized macros (max is <max>)`	Fatal. Pre-processor limit exceeded.

No	*Error message*	*Suggestion*
29	`Too deep macro nesting (max is 100)`	Fatal. Pre-processor limit exceeded.
30	`Actual macro parameter too long (max is 512)`	A single macro (in `#define`) argument may not exceed the length of a source line.
31	`Macro <macro> called with too many parameters`	The number of parameters used was more than the number in the macro declaration.
32	`Macro <macro> called with too few parameters`	The number of parameters used was less than the number in the macro declaration (`#define`).
33	`too many MACRO arguments`	The number of assembler macros exceeds 32.
34	`may not be redefined`	Assembler macros may not be redefined.
35	`no name on macro`	Assembler macro definition without a label was encountered.
36	`Illegal formal parameter in macro`	A parameter that was not an identifier was found.
37	`ENDM or EXITM not in macro`	`ENDM` directive or `EXITM` directive encountered while not inside macro.
38	`'>' expected but found end-of-line`	A < was found but no matching >.
39	`END before start of module`	End-of-module directive has no matching `MODULE` directive.
40	`bad instruction`	The mnemonic/directive does not exist.
41	`bad label`	Labels must begin with `A-Z`, `a-z`, _, or `?`. The succeeding characters must be `A-Z`, `a-z`, `0-9`, _, or `?`. Labels cannot have the same name as a predefined symbol.

No	*Error message*	*Suggestion*
42	`duplicate label`	The label has already appeared in the label field or been declared as `EXTERN`.
43	`illegal effective address`	The addressing mode (operands) is not allowed for this mnemonic.
44	`',' expected`	A comma was expected but not found.
45	`name duplicated`	The name of `RSEG`, `STACK`, or `COMMON` segments is already used but for something else.
46	`segment type expected`	In `RSEG`, `STACK`, or `COMMON` directive `:` was found but the segment type that should follow was not valid.
47	`segment name expected`	The `RSEG`, `STACK`, and `COMMON` directives need a name.
48	`value out of range '<range>'`	The value exceeds its limits.
49	`alignment already set`	`RSEG`, `STACK`, and `COMMON` segment do not allow alignment to be set more than once. Use `ALIGN`, `EVEN`, or `ODD` instead.
50	`undefined symbol: <symbol>`	The symbol did not appear in label field nor in an `EXTERN` or `sfr` declaration.
51	`Can't be both PUBLIC and EXTERN`	Symbols can be declared as either `PUBLIC` or `EXTERN`.
52	`EXTERN not allowed`	Reference to `EXTERN` symbols is not allowed in this context.
53	`expression must be absolute`	The expression cannot involve relocatable or external symbols.

No	*Error message*	*Suggestion*
54	`expression can not be forward`	The assembler must be able to solve the expression the first time this expression is encountered.
55	`illegal size`	The maximum size for expressions is 32 bits.
56	`too many digits`	The value exceeds the size of the destination.
57	`unbalanced conditional assembly directives`	Missing conditional assembly `IF` or `ENDIF`.
58	`ELSE without IF`	Missing conditional assembly `IF`.
59	`ENDIF without IF`	Missing conditional assembly `IF`.
60	`unbalanced structured assembly directives`	Missing structured assembly `IF` or `ENDIF`.
61	`'+' or '-' expected`	Plus or minus sign missing.
62	`Illegal operation on extern or public symbol`	An illegal operation has been used on a public or external symbol; eg `SET`.
63	`Illegal operation on non-constant label`	It is not allowed to make a non-constant symbol `PUBLIC` or `EXTERN`.
64	`Extern or unsolved expression`	The expression must be solved at assembly time, ie not include external references.
65	`'=' expected`	Equals sign was missing.
66	`Segment too long (max is <max>)`	The length of `ASEG`, `RSEG`, `STACK`, or `COMMON` segments is larger than the addressable length.
67	`Public did not appear in label field`	A symbol was declared `PUBLIC` but no label with the same name was found in the source file.
68	`End of block-repeat without start`	The repeat directive `REPT` was not found although the `ENDR` directive was.

No	*Error message*	*Suggestion*
69	`Segment must be relocatable`	The operation is not allowed on `ASEG`.
70	`Limit exceeded: <error text>, value is: <value> (decimal)`	The value exceeded the limits set with the `LIMIT` directive. The `error text` is set by the user in the `LIMIT` directive.
71	`Symbol '<symbol>' has already been declared EXTERN`	An attempt to redeclare an `EXTERN` as `EXTERN` was made.
72	`Symbol '<symbol>' has already been declared PUBLIC`	An attempt to redeclare a `PUBLIC` as `PUBLIC` was made.
73	`End-of-module missing`	A `PROGRAM` or `MODULE` directive was encountered before `ENDMOD` was found.
74	`Expression must yield non-negative result`	The expression was evaluated to a negative number, whereas a positive number was required.
75	`Repeat directive unbalanced`	This error is caused by a `REPT` directive without a matching `ENDR`, or a an `ENDR` directive without a matching `REPT`.
76	`End of repeat directive is missing`	A `REPT` directive without a closing `ENDR` was encountered.
77	`LOCALs not allowed in this context, (<symbol>)`	Local symbols must be declared within macro definitions.
78	`End of macro expected`	An assembler macro is being defined but there was no end-of-macro.
79	`End of repeat expected`	One of the repeat directives is active, but there was no end-of-repeat found.
80	`End of conditional assembly expected`	Conditional assembly is active but there was no end of `if`.

No	*Error message*	*Suggestion*
81	`End of structured assembly expected`	One of the directives for structured assembly is active but has no matching `END`.
82	`Misplaced end of structured assembly`	A directive that terminates one of the structured assembly directives was found but no matching `START` directive is active.
83	`Error in SFR attribute definition`	The attribute being assigned to a label via `SFRTYPE` is illegal.
84	`Illegal symbol type in symbol <symbol>`	The symbol cannot be used in this context since it has the wrong type.
85	`Wrong nr of arguments`	Expected a different number of arguments.
86	`Number expected`	Something else than digits encountered.
87	`Label must be public or extern`	The labels must be declared with `PUBLIC` or `EXTERN`.

H8 ASSEMBLER

In addition to the general errors, the H8 Assembler can generate the following errors:

No	*Error message*	*Suggestion*
400	`Register R0-R7 not allowed here`	
401	`Too many operands`	The instruction could not take as many operands as were specified.
402	`Suffix expected`	A suffix was expected to follow the mnemonic or directive.
403	`Illegal option`	An illegal argument to the `OPT` directive was specified.
404	`Illegal condition code`	

No	*Error message*	*Suggestion*
405	`BREAK illegal.`	
406	`Label must be an SFR`	
407	`'=' or'<>' expected`	
408	`Operand or expression error`	An illegal operand or expression was encountered in the arguments following an instruction.
409	`Structured assembly error.`	
410	`Illegal suffix`	An illegal suffix followed a mnemonic or directive.
411	`Illegal address size`	An illegal address size was specified using *`size`*. For example: `JSR h'1234:17`
412	`Illegal displacement size`	An illegal displacement size was specified using *`size`*. For example: `BSR h'1234:17`
413	`Illegal immediate operand size`	An illegal immediate operand size was specified using *`size`*. For example: `MOV #h'1234:17,R0`
414	`Attempt to read/write a non readable/writeable SFR`	
415	`Register list not allowed for this processor`	Register lists are only allowed for H8S processors.
416	`Illegal register list range`	Wrong range is given for the register list.

No	*Error message*	*Suggestion*
417	`Only stack pointer is allowed`	Another register has been specified for an instruction that is only valid for the stack pointer, ER7, or R7. For example: `STM.L (ER0, ER1),@ER2.`
418	`The immediate value is out of range`	The immediate value is too large or too small; for example: `BSET #9,@ER2.`

XLINK DIAGNOSTICS

This chapter describes the errors and warnings produced by the XLINK Linker.

INTRODUCTION

The error messages produced by the XLINK Linker fall into five categories:

- Linker warning messages.
- Linker error messages.
- Linker fatal error messages.
- Memory overflow message.
- Linker internal error messages.

XLINK WARNING MESSAGES

XLINK warning messages will appear when the linker detects something that may be wrong. The code generated may still be correct.

XLINK ERROR MESSAGES

XLINK error messages are produced when the linker detects something wrong. The linking process will not be aborted but the code produced may be faulty.

XLINK FATAL ERRORS

XLINK fatal error messages abort the linking process. They occur when continued linking is useless, ie the fault is irrecoverable.

MEMORY OVERFLOW MESSAGE

XLINK is a memory-based linker. If run on a system with a small main memory or if very large source files are being used, XLINK may run out of memory. This is recognized by the following message:

```
* * * LINKER OUT OF MEMORY * * *

Dynamic memory used: nnnnnn bytes
```

If this occurs, the solution is either to add system memory, or to enable file bound processing with the `-m` option. The `-t` option can also be used to save memory.

XLINK INTERNAL ERRORS

During linking, a number of internal consistency checks are performed. If any of these checks fail, the linker will terminate after giving a short description of the problem. These errors will not normally occur, but if they do please report them to the IAR Systems technical support group. Please include all possible information about the problem and also a disk with the program that generated the error.

ERROR MESSAGES

If you get a message that indicates a corrupt object file, reassemble or recompile the faulty file since an interrupted assembly or compilation may produce an invalid object file.

The following table lists the XLINK error messages:

No	*Error message*	*Suggestion*
0	`Format chosen cannot support banking`	Format unable to support banking.
1	`Corrupt file. Unexpected end of file in module` *`module`* `(`*`file`*`) encountered`	Linker aborts immediately. Recompile or reassemble, or check the compatibility between the linker and C compiler.
2	`Too many errors encountered (>100)`	Linker aborts immediately.

No	*Error message*	*Suggestion*
3	`Corrupt file. Checksum failed in module` *`module`* `(`*`file`*`). Linker checksum is linkcheck, module checksum is modcheck`	Linker aborts immediately. Recompile or reassemble.
4	`Corrupt file. Zero length identifier encountered in module` *`module`* `(`*`file`*`)`	Linker aborts immediately. Recompile or reassemble.
5	`Address type for CPU incorrect. Error encountered in module` *`module`* `(`*`file`*`)`	Linker aborts immediately. Check that you are using the right files and libraries.
6	`Program module` *`module`* `declared twice, redeclaration in file` *`file`*`. Ignoring second module`	XLINK will not produce code unless the `-B` option (forced dump) is used.
7	`Corrupt file. Unexpected UBROF - format end of file encountered in module` *`module`* `(`*`file`*`)`	Linker aborts immediately. Recompile or reassemble.
8	`Corrupt file. Unknown or misplaced tag encountered in module` *`module`* `(`*`file`*`). Tag` *`tag`*	Linker aborts immediately. Recompile or reassemble.
9	`Corrupt file. Module` *`module`* `start unexpected in file` *`file`*	Linker aborts immediately. Recompile or reassemble.
10	`Corrupt file. Segment no.` *`segno`* `declared twice in module` *`module`* `(`*`file`*`)`	Linker aborts immediately. Recompile or reassemble.
11	`Corrupt file. External no.` *`ext no`* `declared twice in module` *`module`* `(`*`file`*`)`	Linker aborts immediately. Recompile or reassemble.

No	*Error message*	*Suggestion*
12	`Unable to open file` *`file`*	Linker aborts immediately. If you are using the command line check the environment variable `XLINK_DFLTDIR`.
13	`Corrupt file. Error tag encountered in module` *`module`* `(`*`file`*`)`	A UBROF error tag was encountered. Linker aborts immediately. Recompile or reassemble.
14	`Corrupt file. Local` *`local`* `defined twice in module` *`module`* `(`*`file`*`)`	Linker aborts immediately. Recompile or reassemble.
15	`Faulty bank definition -bbank def`	Incorrect syntax. Linker aborts immediately.
16	`Segment` *`segment`* `is too long for segment definition`	The segment defined does not fit into the memory area reserved for it. Linker aborts immediately.
17	`Segment` *`segment`* `is defined twice in segment definition -Zsegdef`	Linker aborts immediately.
18	`Range error in module` *`module`* `(`*`file`*`), segment` *`segment`* `at address address. Value` *`value`*`, in tag` *`tag`*`, is out of bounds`	The address is out of the CPU address range. Locate the cause of the problem using the information given in the error message.
	The check can be suppressed by the `-R` option.	
19	`Corrupt file. Undefined segment referenced in module` *`module`* `(`*`file`*`)`	Linker aborts immediately. Recompile or reassemble.
20	`Undefined external referenced in module` *`module`* `(`*`file`*`)`	Linker aborts immediately. Recompile or reassemble.
21	`Segment` *`segment`* `in module` *`module`* `does not fit bank`	The segment is too long. Linker aborts immediately.

No	*Error message*	*Suggestion*
22	`Paragraph no. is not applicable for the wanted CPU. Tag encountered in module module (file)`	Linker aborts immediately. Delete the paragraph no. declaration in the `.xcl` file.
23	`Corrupt file. T_REL_FI_8 or T_EXT_FI_8 is corrupt in module module (file)`	The tag `T_REL_FI_8` or `T_EXT_FI_8` is faulty. Linker aborts immediately. Recompile or reassemble.
24	`Segment segment overlaps segment segment`	The segments overlap each other; ie both have code on the same address.
25	`Corrupt file. Unable to find module module (file)`	A module is missing. Linker aborts immediately.
26	`Segment segment is too long`	This error should never occur unless the program is extremely large. Linker aborts immediately.
27	`Entry entry in module module (file) redefined in module module (file)`	There are two or more entries with the same name. Linker aborts immediately.
28	`File file is too long`	The program is too large. Split the file. Linker aborts immediately.
29	`No object file specified in command-line`	There is nothing to link. Linker aborts immediately.
30	`Option -option also requires the -option option`	Linker aborts immediately.
31	`Option -option cannot be combined with the -option option`	Linker aborts immediately.
32	`Option -option cannot be combined with the -option option and the -option option`	Linker aborts immediately.

No	*Error message*	*Suggestion*
33	`Faulty value val (in command line or in XLINK_PAGE), (range is 10-150)`	Faulty page setting. Linker aborts immediately.
34	`Filename too long`	The filename is more than 255 characters long. Linker aborts immediately.
35	`Unknown flag flag in cross reference option option`	Linker aborts immediately.
36	`Option op does not exist`	Linker aborts immediately.
37	`- not succeeded by character`	The - marks the beginning of an option, and must be followed by a character. Linker aborts immediately.
38	`Option option multiply defined`	Linker aborts immediately.
39	`Illegal character specified in option op`	Linker aborts immediately.
40	`Argument expected after option op`	This option must be succeeded by an argument. Linker aborts immediately.
41	`Unexpected '-' in option op`	Linker aborts immediately.
42	`Faulty symbol definition -Dsymbol definition`	Incorrect syntax. Linker aborts immediately.
43	`Symbol in symbol definition too long`	The symbol name is more than 255 characters. Linker aborts immediately.
44	`Faulty value val (in command line or in XLINK_COLUMNS), (range 80-300)`	Faulty column setting. Linker aborts immediately.

No	*Error message*	*Suggestion*
45	`Unknown CPU` *`CPU`* `encountered in command line (or in XLINK_CPU)`	Linker aborts immediately. Check the argument to `-c` is valid. If you are using the command line you can get a list of CPUs by typing `xlink` ↵.
46	`Undefined external` *`external`* `referred in module (`*`file`*`)`	Entry to external is missing.
47	`Unknown format` *`format`* `encountered in command line or XLINK_FORMAT`	Linker aborts immediately.
48	`Faulty segment definition -Zsegdef`	Incorrect syntax. Linker aborts immediately.
49	`Segment name in segment definition too long`	255 characters long. Linker aborts immediately.
50	`Paragraph no. not allowed for this CPU, encountered in option` *`option`*	Linker aborts immediately. Do not use paragraph no. in declarations.
51	`Hexadecimal or decimal value expected in option` *`option`*	Linker aborts immediately.
52	`Overflow on value in option` *`option`*	Linker aborts immediately.
53	`Parameter exceeded 255 characters in extended command line file` *`file`*	Linker aborts immediately.
54	`Extended command line file` *`file`* `is empty`	Linker aborts immediately.
55	`Extended command line variable XLINK_ENVPAR is empty`	Linker aborts immediately.
56	`Overlapping ranges in segment definition` *`segment def`*	Linker aborts immediately.

No	*Error message*	*Suggestion*
57	`No CPU defined`	No CPU defined, either in the command line or in `XLINK_CPU`. Linker aborts immediately.
58	`No format defined`	No format defined, either in the command line or in `XLINK_FORMAT`. Linker aborts immediately.
59	`Revision no. for file is incompatible with XLINK revision no.`	Linker aborts immediately. If this error occurs after recompilation or reassembly, the wrong version of XLINK is being used. Check with your supplier.
60	`Segment` *`segment`* `defined in bank definition and segment definition.`	Linker aborts immediately.
61	`Symbol in bank definition is too long`	Linker aborts immediately.
62	`File` *`file`* `multiply defined in command line`	Linker aborts immediately.
63	`Trying to pop an empty stack in module` *`module`* `(`*`file`*`)`	Linker aborts immediately. Recompile or reassemble.
64	`Module` *`module`* `(`*`file`*`) has not the same debug type as the other modules`	Linker aborts immediately.
65	`Faulty replacement definition -r`*`replacement`* `definition`	Incorrect syntax. Linker aborts immediately.
66	`Function with F-index` *`index`* `has not been defined before indirect reference in module` *`module`* `(`*`file`*`)`	Indirect call to an undefined in module. Probably caused by an omitted function declaration.

No	*Error message*	*Suggestion*
67	`Function` *`name`* `has same F-index as function-`*`name`*`, defined in module` *`module`* `(`*`file`*`)`	Probably a corrupt file. Recompile file.
68	`External function` *`name`* `in module` *`module`* `(`*`file`*`) has no global definition`	If no other errors have been encountered, this error is generated by an assembly language call from C where the required declaration using the `$DEFFN` assembly language support directive is missing. The declaration is necessary to inform the linker of the memory requirements of the function.
69	`Indirect or recursive function` *`name`* `in module` *`module`* `(`*`file`*`) has parameters or auto variables in nondefault memory`	The recursively or indirectly called function *`name`* is using extended language memory specifiers (`bit`, `data`, `idata`, etc) to point to non-default memory, which is not allowed. Function parameters to indirectly called functions must be in the default memory area for the memory model in use, and for recursive functions, both local variables and parameters must be in default memory.
70	`Module` *`module`* `(`*`file`*`) has not the same memory as previously linked modules`	Only modules compiled under the same memory model may be linked together.
71	`Segment` *`name`* `is incorrectly defined (in a bank definition, has wrong segment type or mixed segment types)`	This is usually due to misuse of a predefined segment; see the explanation of *`name`* in the *H8 C Compiler Programming Guide*. It may be caused by changing the predefined linker control file.

No	*Error message*	*Suggestion*
72	`Segment name must be defined in a -Z definition`	This is either an omission of a segment in the linker (usually a segment needed by the C system control) file or a spelling error (segment names are case sensitive).
73	`Label ?ARG_MOVE not found (recursive function need it)`	In the library there should be a module containing this label. If it has been removed it must be restored.
74	`There was an error when writing to file file`	Either the linker or your host system is corrupt, or the two are incompatible.
75	`SFR address in module module (file), segment segment at address address, value value is out of bounds`	An `SFR` has been defined to a bad address. Change the definition.
76	`Absolute segments overlap in module module`	The linker has found two or more absolute segments in *`module`* overlapping each other.
77	`Absolute segments in module module (file) overlaps absolute segment in module module (file)`	The linker has found two or more absolute segments in *`module`*`(`*`file`*`)` and *`module`* `(`*`file`*`)` overlapping each other.
78	`Absolute segment in module module (file) overlaps segment segment`	The linker has found an absolute segment in *`module`*`(`*`file`*`)` overlapping a relocatable segment.
79	`Faulty allocation definition -adefinition`	The linker has discovered an error in an overlay control definition.
80	`Symbol in allocation definition (-a) too long`	A symbol in the `-a` command is too long.

No	*Error message*	*Suggestion*
81	`Unknown flag in extended format option -Y`	Check flags.
82	`Conflict in segment 'name'. Mixing overlayable and not overlayable segment parts.`	These errors only occur with the 8051 and converted PL/M code.
83	`The overlayable segment 'name' may not be banked.`	These errors only occur with the 8051 and converted PL/M code.
84	`The overlayable segment 'name' must be of relative type.`	These errors only occur with the 8051 and converted PL/M code.

WARNING MESSAGES

The following table lists the linker warning messages:

No	*Warning message*	*Suggestion*
0	`Too many warnings`	Too many warnings encountered.
1	`Error tag encountered in module` *`module`* `(`*`file`*`)`	A UBROF error tag was encountered when loading file *`file`*. This indicates a corrupt file and will generate an error in the linking phase.
2	`Symbol` *`symbol`* `is redefined in command-line`	A symbol has been redefined.
3	`Type conflict. Segment` *`segment`*`, in module` *`module`*`, is incompatible with earlier segment(s) of the same name`	Segments of the same name should have the same type.

No	*Warning message*	*Suggestion*
4	`Close/open conflict. Segment` *`segment`*`, in module` *`module`*`, is incompatible with earlier segment of the same name`	Segments of the same name should be either open or closed.
5	`Segment` *`segment`* `cannot be combined with previous segment`	The segments will not be combined.
6	`Type conflict for external/entry` *`entry`*`, in module` *`module`*`, against external/entry in module` *`module`*	Entries and their corresponding externals should have the same type.
7	`Module` *`module`* `declared twice, once as program and once as library. Redeclared in file` *`file`*`, ignoring library module`	The program module is linked.
8	`Segment` *`segment`* `undefined in segment or bank definition`	Undefined segment exists. All segments should be defined in either the segment or the bank definition.
9	`Ignoring redeclared program entry`	Only the program entry found first is chosen.
10	`No modules to link`	The linker has no modules to link.
11	`Module` *`module`* `declared twice as library. Redeclared in file` *`file`*`, ignoring second module`	The module found first is linked.
12	`Using SFB in banked segment` *`segment`* `in module` *`module`* `(`*`file`*`)`	The `SFB` assembler directive may not work in a banked segment.

No	*Warning message*	*Suggestion*
13	`Using SFE in banked segment` *`segment`* `in module` *`module`* `(`*`file`*`)`	The `SFE` assembler directive may not work in a banked segment.
14	`Entry` *`entry`* `duplicated. Module` *`module`* `(`*`file`*`) loaded, module` *`module`* `(`*`file`*`) discarded`	Duplicated entries exist in conditionally loaded modules; ie library modules or conditionally loaded program modules (with the `-C` option).
15	`Predefined type sizing mismatch between modules` *`module`* `(`*`file`*`) and` *`module`* `(`*`file`*`)`	The modules have been compiled with different options for predefined types, such as different sizes of basic C types (eg `integer`, `double`).
16	`Function` *`name`* `in module` *`module`* `(`*`file`*`) is called from two function trees (with roots` *`name1`* `and` *`name2`*`)`	The probable cause is that an interrupt function calls another function that also could be executed by a foreground program, and this could lead to execution errors.
17	`Segment` *`name`* `is too large or placed at wrong address`	This error occurs if a given segment overruns the available address space in the named memory area. To find out the extent of the overrun do a dummy link, moving the start address of the named segment to the lowest address, and look at the linker map file. Then relink with the correct address specification.
18	`Segment` *`segment`* `overlaps segment` *`segment`*	The linker has found two relocatable segments overlapping each other. Check the `-Z` option parameters.

No	*Warning message*	*Suggestion*
19	`Absolute segments overlaps in module` *`module`* `(`*`file`*`)`	The linker has found two or more absolute segments in module *`module`* overlapping each other.
20	`Absolute segment in module` *`module`* `(`*`file`*`) overlaps absolute segment in module` *`module`* `(`*`file`*`)`	The linker has found two or more absolute segments in module *`module`* `(`*`file`*`)` and module *`module`* `(`*`file`*`)` overlapping each other. Change the `ORG` directives.
21	`Absolute segment in module` *`module`* `(`*`file`*`) overlaps segment` *`segment`*	The linker has found an absolute segment in module *`module`* `(`*`file`*`)` overlapping a relocatable segment. Change either the `ORG` directive or the `-Z` relocation command.
22	`Interrupt function` *`name`* `in module` *`module`* `(`*`file`*`) is called from other functions`	Interrupt functions may not be called.

XLIB DIAGNOSTICS

This chapter lists the messages produced by the XLIB Librarian.

XLIB MESSAGES

The following table lists the XLIB messages. Commands flagged as erroneous never alter object files.

No	*Error message*	*Suggestion*
1	`Bad object file, EOF encountered`	Bad or empty object file, which could be the result of an aborted assembly or compilation.
2	`Unexpected EOF in batch file`	The last command in a command file must be `EXIT`.
3	`Unable to open file` *`file`*	Could not open the command file or, if `ON-ERROR-EXIT` has been specified, this message is issued on any failure to open a file.
4	`Variable length record out of bounds`	Bad object module, could be the result of an aborted assembly.
5	`Missing or non-default parameter`	A parameter was missing in the direct mode.
6	`No such CPU`	A list with the possible choices is displayed when this error is found.
7	`CPU undefined`	`DEFINE-CPU` must be issued before object file operations can begin. A list with the possible choices is displayed when this error is found.
8	`Ambiguous CPU type`	A list with the possible choices is displayed when this error is found.
9	`No such command`	Use the `HELP` command.

No	*Error message*	*Suggestion*
10	`Ambiguous command`	Use the `HELP` command.
11	`Invalid parameter(s)`	Too many parameters or a misspelled parameter.
12	`Module out of sequence`	Bad object module, could be the result of an aborted assembly.
13	`Incompatible object, consult distributor!`	Bad object module, could be the result of an aborted assembly, or that the assembler/compiler revision used is incompatible with the version of XLIB used.
14	`Unknown tag: hh`	Bad object module, could be the result of an aborted assembly.
15	`Too many errors`	More than 32 errors will make XLIB abort.
16	`Assembly/compilation error?`	The `T_ERROR` tag was found. Edit and re-assemble/re-compile your program.
17	`Bad CRC, hhhh expected`	Bad object module; could be the result of an aborted assembly.
18	`Can't find module: xxxxx`	Check the available modules with `LIST-MOD` *`file`*.
19	`Module expression out of range`	Module expression is less than one or greater than `$`.
20	`Bad syntax in module expression: xxxxx`	The syntax is invalid.
21	`Illegal insert sequence`	The specified *`destination`* in the `INSERT-MODULES` command must not be within the *`start-end`* sequence.
22	`<End module> found before <Start module>!`	Source module range must be from low to high order.
23	`Before or after!`	Bad `BEFORE/AFTER` specifier in the `INSERT-MODULES` command.

No	*Error message*	*Suggestion*
24	`Corrupt file, error occurred in` *`tag`*	A fault is detected in the object file *`tag`*. Reassembly or recompilation may help. Otherwise contact your supplier.
25	*`File`* `is write protected`	The file *`file`* is write protected and cannot be written to.
26	`Non-matching replacement module` *`name`* `found in source file`	In the source file, a module name with no corresponding entry in the destination file was found.

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

SYMBOLS