



H8 C COMPILER

Programming Guide

COMMAND LINE VERSION

COPYRIGHT NOTICE

© Copyright 1996 IAR Systems. All rights reserved.

No part of this document may be reproduced without the prior written consent of IAR Systems. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

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: ICCH8C-1

This documentation was produced by Human-Computer Interface.

WELCOME

Welcome to the H8 C Compiler Programming Guide.

This guide provides reference information about the IAR Systems C Compiler 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 Assembler refer to the *H8 Assembler, Linker, and Librarian 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 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.

The *Introduction* provides a brief summary of the H8 C Compiler's features.

The *Tutorial* illustrates how you might use the C compiler to develop a series of typical programs, and illustrates some of the compiler's most important features. It also describes a typical development cycle using the C compiler.

C compiler options summary explains how to set the C compiler options, and gives a summary of them.

C compiler options reference gives information about each C compiler option.

Configuration then describes how to configure the C compiler for different requirements.

Data representation describes how the compiler represents each of the C data types and gives recommendations for efficient coding.

General C library definitions gives an introduction to the C library functions, and summarizes them according to header file.

C library functions reference then gives reference information about each library function.

Language extensions summarizes the extended keywords, `#pragma` keywords, predefined symbols, and intrinsic functions specific to the H8 C Compiler.

Extended keyword reference then gives reference information about each of the extended keywords.

#pragma directive reference gives reference information about the `#pragma` keywords.

Predefined symbols reference gives reference information about the predefined symbols.

Intrinsic function reference gives reference information about the intrinsic functions.

Assembly language interface describes the interface between C programs and assembly language routines.

Segment reference gives reference information about the C compiler's use of segments.

K&R and ANSI C language definitions describes the differences between the K&R description of the C language and the ANSI standard.

Diagnostics lists the compiler warning and error messages.

ASSUMPTIONS

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

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

This guide does not attempt to describe the C language itself. For a description of the C language, *The C Programming Language* by Kernighan and Richie is recommended, of which the latest edition also covers ANSI C.

CONVENTIONS

This guide uses the following typographical conventions:

<i>Style</i>	<i>Used for</i>
computer	Text that you type in, or that appears on the screen.
<i>parameter</i>	A label representing the actual value you should type as part of a command.
[<i>option</i>]	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.
<i>reference</i>	A cross-reference to another part of this guide, or to another guide.

In this guide K&R is used as an abbreviation for *The C Programming Language* by Kernighan and Richie.

PREFACE

CONTENTS

INSTALLATION AND DOCUMENTATION ROUTE MAP	1
Command line versions	1
Windows Workbench versions	2
UNIX versions	3
Documentation route map	4
INTRODUCTION	5
C compiler	5
TUTORIAL	7
Typical development cycle	8
Getting started	9
Creating a program	10
Extending the program	22
Adding an interrupt handler	27
C COMPILER OPTIONS SUMMARY	31
Setting C compiler options	31
Options summary	32
C COMPILER OPTIONS REFERENCE	35
Code generation	35
Debug	44
#define	45
List	46
#undef	52
Include	53
Target	54
Miscellaneous	55
CONFIGURATION	61
Introduction	61
Processor group	62
XLINK command file	62
Run-time library	63
Memory model	64
Floating-point precision	64

Stack size	65
Input and output	66
Register I/O	69
Heap size	69
Initialization	69
DATA REPRESENTATION	73
Data types	73
Pointers	76
Efficient coding	77
GENERAL C LIBRARY DEFINITIONS	79
Introduction	79
C LIBRARY FUNCTIONS REFERENCE	87
LANGUAGE EXTENSIONS	157
Introduction	157
Extended keywords summary	157
#pragma directive summary	158
Predefined symbols summary	159
Intrinsic function summary	160
Other extensions	162
EXTENDED KEYWORD REFERENCE	163
#PRAGMA DIRECTIVE REFERENCE	175
PREDEFINED SYMBOLS REFERENCE	189
INTRINSIC FUNCTION REFERENCE	193
ASSEMBLY LANGUAGE INTERFACE	207
Creating a shell	207
Calling convention	208
Calling assembly routines from C	210
SEGMENT REFERENCE	211

K&R AND ANSI C LANGUAGE DEFINITIONS.....	219
Introduction	219
Definitions	219
DIAGNOSTICS.....	225
Compilation error messages	227
Compilation warning messages	243
INDEX	253

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).

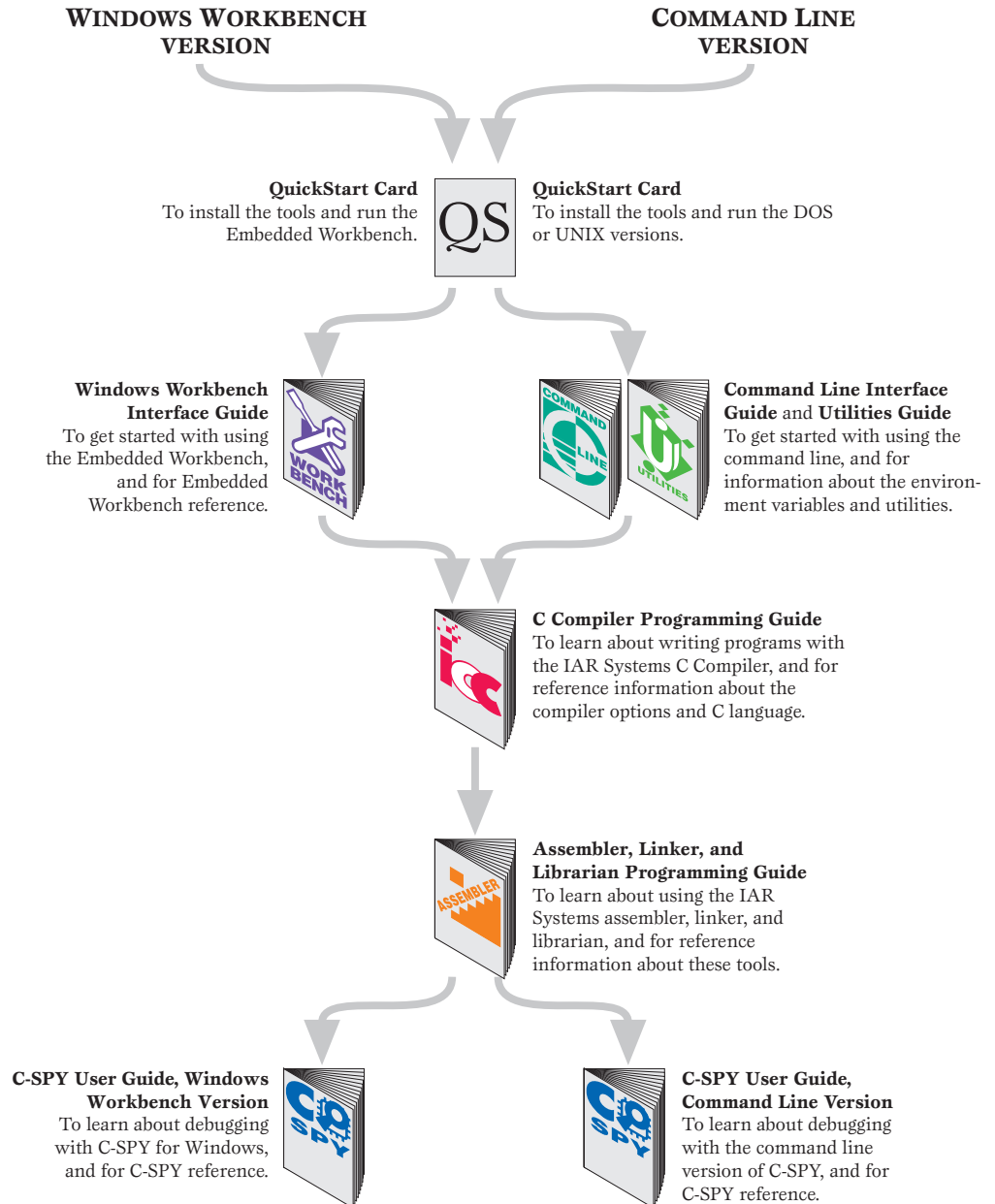
INSTALLATION

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 C Compiler, and provides information about running it from the command line.

C COMPILER

The IAR Systems C Compiler for the H8 family of microprocessors offers the standard features of the C language, plus many extensions designed to take advantage of the H8-specific facilities. The compiler is supplied with the IAR Systems Assembler for the H8, with which it is integrated, and shares linker and librarian manager tools.

It provides the following features:

LANGUAGE FACILITIES

- ◆ Conformance to the ANSI specification.
- ◆ Standard library of functions applicable to embedded systems, with source optionally available.
- ◆ IEEE-compatible floating-point arithmetic.
- ◆ Powerful extensions for H8-specific features, including efficient I/O.
- ◆ Linkage of user code with assembly routines.
- ◆ Long identifiers – up to 255 significant characters.

PERFORMANCE

- ◆ Fast compilation.
- ◆ Memory-based design which avoids temporary files or overlays.
- ◆ Rigorous type checking at compile time.
- ◆ Rigorous module interface type checking at link time.
- ◆ LINT-like checking of program source.

CODE GENERATION

- ◆ Selectable optimization for code speed or size.
- ◆ Comprehensive output options, including relocatable binary, ASM, ASM + C, XREF, etc.
- ◆ Easy-to-understand error and warning messages.
- ◆ Compatibility with the C-SPY high-level debugger.

TARGET SUPPORT

- ◆ Small and large memory models.
- ◆ Flexible variable allocation.
- ◆ Interrupt functions requiring no assembly language.
- ◆ A `#pragma` directive to maintain portability while using processor-specific extensions.

TUTORIAL

This chapter illustrates how you might use the H8 C Compiler to develop a series of typical programs, and illustrates some of the C compiler's most important features:

Before reading this chapter you should:

- ◆ Have installed the C compiler 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 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.

Summary of tutorial files

The following table summarizes the tutorial files used in this chapter:

<i>File</i>	<i>What it demonstrates</i>
tutor1	Compiling and running a simple C program.
tutor2	Using serial I/O.
tutor3	Interrupt handling.

RUNNING THE EXAMPLE PROGRAMS

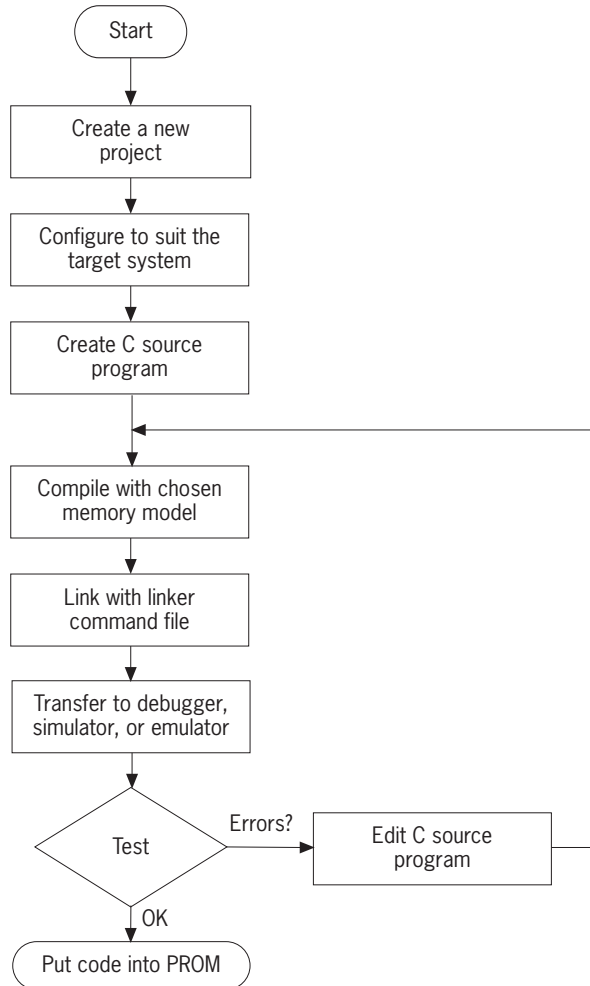
This tutorial shows how to run the example programs using the optional C-SPY simulator.

You can also run the examples on a target system with an EPROM emulator and debugger. In this case you will first need to configure the I/O routines.

Alternatively, you may still follow this tutorial by examining the list files created. The `.lst` and `.map` files show which areas of memory to monitor.

**TYPICAL
DEVELOPMENT CYCLE**

Development will normally follow the cycle illustrated below:



The following tutorial follows this cycle.

GETTING STARTED

The first step in developing a project using the C compiler is to decide on an appropriate configuration to suit your target system.

CONFIGURING TO SUIT THE TARGET SYSTEM

Our tutorial programs contain only a small amount of code, and so only require the small memory model. We will use the chip option `-v0`, which generates code for the H8/300H processor.

Each project needs an XLINK command file containing details of the target system's memory map.

Choosing the linker command file

A suitable linker command file for the small memory model, `lnkh8hs.xcl`, is provided in the `icch8` subdirectory.

Examine `lnkh8hs.xcl` using a suitable text editor, such as the MS-DOS `edit` editor.

The file first contains the following XLINK command to define the CPU type as H8/300H small mode (`-ms`):

```
-ch8
```

It then contains a series of `-Z` commands to define the segments used by the compiler. The key segments are as follows:

<i>Segment type</i>	<i>Segment names</i>	<i>Address range</i>
CODE	INTVEC, IFLIST, FLIST	0x00 to 0xFF
CODE	CDATA0, CDATA1, RCODE CODE, CDATA2, CDATA3 CONST, CSTR, CCSTR	0x0100 to 0xBFFF
DATA	IDATA0, UDATA0	0xFF00 to 0xFF0F
DATA	IDATA1, UDATA1, IDATA2 IDATA3, UDATA2, UDATA3 ECSTR, WCSTR, TEMP	0xC000 to 0xFEFF
DATA	CSTACK	After DATA, size 0x200
DATA	NO_INIT	START-END
BIT	BITVARS	0xFB00 bit 0 to 0xFF80 bit 7

The file defines the routines to be used for `printf`, `sprintf`, `scanf`, and `sscanf`.

Finally it contains the following line to load the appropriate C library:

```
clh8hs
```

See *Run-time library*, page 63, for details of the different C libraries provided.

Note that these definitions are not permanent: they can be altered later on to suit your project if the original choice proves to be incorrect, or less than optimal.

For detailed information on configuring to suit the target memory, see *Memory location*, page 64. For detailed information on choosing stack size, see *Stack size*, page 65.

CREATING A NEW PROJECT

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

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 `icch8` directory. Select this directory by entering the command:

```
cd c:\iar\icch8 ↵
```

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

CREATING A PROGRAM The first tutorial demonstrates how to compile, link, and run a program.

ENTERING THE PROGRAM

The first program is a simple program using only standard C facilities. It repeatedly calls a function that increments a variable:

```
#include <stdio.h>
int call_count;
unsigned char my_char;
const char con_char='a';

void do_foreground_process(void)
```

```

{
    call_count++;
    putchar(my_char);
}

void main(void)
{
    int my_int=0;
    call_count=0;
    my_char=con_char;
    while (my_int<100)
    {
        do_foreground_process();
        my_int++;
    }
}

```

Enter the program using any standard text editor, such as the MS-DOS edit editor, and save it in a file called `tutor1.c`. Alternatively, a copy is provided in the C compiler files directory.

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

COMPILING THE PROGRAM

To compile the program enter the command:

```
icch8 tutor1 -v0 -ms -r -L -e -q -I\iar\inc ↵
```

There are several compile options used here:

<i>Option</i>	<i>Description</i>
-v0	Selects the H8/300H processor.
-ms	Selects the small memory model.
-r	Allows the code to be debugged with C-SPY.
-L	Creates a list file.
-e	Enables extended commands (not needed in this tutorial).
-q	Includes assembler code with C in the listing.
-I	Specifies the pathname for include files.

This creates an object module called `tutor1.r37` and a list file called `tutor1.lst`.

Viewing the listing

Examine the list file produced and see how the variables are assigned to different segments.

```
#####
#
# IAR H8 C-Compiler Vx.xx
# Front End Vx.xx
# Global Optimizer Vx.xx
#
# Target option = H8
# Memory model = small
# Source file = tutor1.c
# List file = tutor1.lst
# Object file = tutor1.r37
# Command line = tutor1 -v0 -ms -r -L -q
#
# (c) Copyright IAR Systems 1996 #
#####

\ 0000 NAME tutor1(16)
\ 0000 RSEG CODE(1)
\ 0000 RSEG CONST(1)
\ 0000 RSEG UDATA1(1)
\ 0000 PUBLIC call_count
\ 0000 PUBLIC con_char
\ 0000 PUBLIC do_foreground_process
\ 0000 PUBLIC main
\ 0000 PUBLIC my_char
\ 0000 EXTERN putchar
\ 0000 EXTERN ?CLH8HS_0_90_L00
\ 0000 RSEG CODE
\ 0000 do_foreground_process:
1 #include <stdio.h>
2 int call_count;
3 unsigned char my_char;
4 const char con_char='a';
5
6 void do_foreground_process(void)
```

```

7      {
8      call_count++;
\ 0000 6B0E0000      MOV.W  @call_count,E6
\ 0004 791E0001      ADD.W  #1,E6
\ 0008 6B8E0000      MOV.W  E6,@call_count
9      putchar(my_char);
\ 000C 6A0E0002      MOV.B  @my_char,R6L
\ 0010 1756          EXTU.W R6
\ 0012 5E000000      JSR   @putchar
10     }
\ 0016 5470          RTS
\ 0018          main:
11
12     void main(void)
13     {
14     int my_int=0;
\ 0018 1988          SUB.W  E0,E0
15     call_count=0;
\ 001A 19EE          SUB.W  E6,E6
\ 001C 6B8E0000      MOV.W  E6,@call_count
16     my_char=con_char;
\ 0020 6A0E0000      MOV.B  @con_char,R6L
\ 0024 6A8E0002      MOV.B  R6L,@my_char
\ 0028          ?0001:          ; [WHILE_CONTINUE]      2:1
17     while (my_int<100)
\ 0028 79280064      CMP.W  #100,E0
\ 002C 4C0A          BGE   ?0000
\ 002E          ?0002:          ; [IF_TRUE]      3:1
18     {
19     do_foreground_process();
\ 002E 5E000000      JSR   @do_foreground_process
20     my_int++;
\ 0032 79180001      ADD.W  #1,E0
21     }
22     }
\ 0036 40F0          BT     ?0001
\ 0038          ?0000:          ; [WHILE_BREAK] 4:1
\ 0038 5470          RTS
\ 0000          RSEG  CONST
\ 0000          con_char:
\ 0000 61          DC.B  'a'

```

```
\ 0000          RSEG  UDATA1
\ 0000      call_count:
\ 0002          DS.B   2
\ 0002      my_char:
\ 0003          DS.B   1
\ 0003          END
```

```
Errors: none
Warnings: none
Code size: 58
Constant size: 1
Static variable size: 3
```

LINKING THE PROGRAM

To link the object file with the appropriate library module to produce code that can be executed by the C-SPY debugger, enter the command:

```
xlink tutor1 -f lnkh8hs -r -x -l tutor1.map ↵
```

The `-f` option specifies your XLINK command file `lnkh8hs`, and the `-r` option allows the code to be debugged with C-SPY.

The `-x` creates a map file and the `-l filename` gives the name of the file.

The result of linking is a code file called `aout.a37` and a map file called `tutor1.map`.

Viewing the map file

Examine the map file to see how the segment definitions and code were placed into their physical addresses. The main points of the map file are shown on the following listing:


```

#####
#
#   IAR Universal Linker Vx.xx
#
#   Target CPU    = h8
#   List file     = tutor1.map
#   Output file 1 = aout.d37
#   Output format = debug
# Command line = tutor1 -f lnkh8s (-ch8
#               -Z(CODE)INTVEC,IFLIST,FLIST=0-FF
#               -Z(CODE)CDATA0,CDATA1,RCODE,CODE,CDATA2,CDATA3,
#               CONST,CSTR,CCSTR=100-BFFF
#               -Z(DATA)IDATA0,UDATA0=FF00-FF0F
#               -Z(DATA)IDATA1,UDATA1, IDATA2, IDATA3,UDATA2,
#               UDATA3,ECSTR,WCSTR,TEMP,CSTACK+200=C000-FAFF
#               -Z(BIT)BITVARS=0-3FF
#               -e_medium_write=_formatted_write
#               -e_medium_read=_formatted_read c1h8s -FMOTOROLA)
#               -r -x -l tutor1.map
#
#                                     (c) Copyright IAR Systems 1996 #
#####

```

Command line
Equivalent command line.

Included XCL file
Commands included in the linker
command file.

```

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

```

Program entry ————— Program entry at : 0100 Relocatable, from module : CSTARTUP
Shows the address of the program entry
point.

```

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

```

Module map —————
Information about each module that was
loaded as part of the program.

File name
Shows the name of the file from which modules were linked.

Module
Type and name.

Segments in the module
A list of the segments in the specified module, with information about each segment.

Entries
Global symbols declared within the segment.

```

FILE NAME : tutor1.r37
PROGRAM MODULE, NAME : tutor1

SEGMENTS IN THE MODULE
=====
CODE
  Relative segment, address : 0172 - 01AB
  ENTRIES                ADDRESS
  do_foreground_process  0172
  main                   018A
  LOCALS                 ADDRESS
  ?0001                  019A
  ?0002                  01A0
  ?0000                  01AA
  -----
CONST
  Relative segment, address : 01B0 - 01B0
  ENTRIES                ADDRESS
  con_char                01B0
  -----
UDATA1
  Relative segment, address : C000 - C002
  ENTRIES                ADDRESS
  call_count              C000
  my_char                 C002
  -----
REF BY MODULE
Not referred to
CSTARTUP
Not referred to
Not referred to
Not referred to
*****

```

Next file

```

FILE NAME : c1h8hs.r37
LIBRARY MODULE, NAME : putchar

SEGMENTS IN THE MODULE
=====
CODE
  Relative segment, address : 01AC - 01AD
  ENTRIES                ADDRESS
  putchar                01AC
  -----
REF BY MODULE
tutor1
LIBRARY MODULE, NAME : ?LIB_VERSION_L00

```

SEGMENTS IN THE MODULE

```

-----
RCODE
  Relative segment, address : Not in use
      ENTRIES                ADDRESS                REF BY MODULE
      ?CLH8HS_0_90_L00      0100                tutor1
-----

```

PROGRAM MODULE, NAME : CSTARTUP

SEGMENTS IN THE MODULE

```

-----
CODE
  Relative segment, address : Not in use
-----
CCSTR
  Relative segment, address : Not in use
-----
TEMP
  Relative segment, address : Not in use
-----
ECSTR
  Relative segment, address : Not in use
-----
CONST
  Relative segment, address : Not in use
-----
CSTR
  Relative segment, address : Not in use
-----
CDATA3
  Relative segment, address : Not in use
-----
CDATA2
  Relative segment, address : Not in use
-----
CDATA1
  Relative segment, address : Not in use
-----
CDATA0
  Relative segment, address : Not in use
-----

```

```
IDATA3
  Relative segment, address : Not in use
  -----
IDATA2
  Relative segment, address : Not in use
  -----
IDATA1
  Relative segment, address : Not in use
  -----
IDATA0
  Relative segment, address : Not in use
  -----
UDATA3
  Relative segment, address : Not in use
  -----
UDATA2
  Relative segment, address : Not in use
  -----
UDATA1
  Relative segment, address : Not in use
  -----
UDATA0
  Relative segment, address : Not in use
  -----
RCODE
  Relative segment, address : 0100 - 0171
  -----
CSTACK
  Relative segment, address : Not in use
  -----
INTVEC
  Common segment, address : 0000 - 0007
  -----
```

Next module — LIBRARY MODULE, NAME : exit

Information about the next module in the current file.

SEGMENTS IN THE MODULE

CODE

Relative segment, address : 01AE - 01AF

ENTRIES	ADDRESS	REF BY MODULE
?C_EXIT	01AE	Not referred to
exit	01AE	CSTARTUP

* * *

* SEGMENTS IN DUMP ORDER *

* * *

Segments in dump order

Lists all the segments that make up the program, in the order linked.

SEGMENT	START ADDRESS	END ADDRESS	TYPE	ORG	P/N	ALIGN
BITVARS	Not in use		dse	stc	pos	0
INTVEC	0000	0007	com	stc	pos	1
IFLIST	Not in use		dse	flt	pos	0
FLIST	Not in use		dse	flt	pos	0
CDATA0	Not in use		rel	stc	pos	1
CDATA1	Not in use		rel	flt	pos	1
RCODE	0100	0171	rel	flt	pos	1
CODE	0172	01AF	rel	flt	pos	1
CDATA2	Not in use		rel	flt	pos	1
CDATA3	Not in use		rel	flt	pos	1
CONST	01B0	01B1	rel	flt	pos	1
CSTR	Not in use		rel	flt	pos	1
CCSTR	Not in use		rel	flt	pos	1
IDATA0	Not in use		rel	stc	pos	1
UDATA0	Not in use		rel	flt	pos	1
IDATA1	Not in use		rel	stc	pos	1
UDATA1	C000	C003	rel	flt	pos	1
IDATA2	Not in use		rel	flt	pos	1
IDATA3	Not in use		rel	flt	pos	1
UDATA2	Not in use		rel	flt	pos	1
UDATA3	Not in use		rel	flt	pos	1
ECSTR	Not in use		rel	flt	pos	1
WCSTR	Not in use		dse	flt	pos	0
TEMP	Not in use		rel	flt	pos	1
CSTACK	C004	C203	rel	flt	pos	1

```
*****
*
*           END OF CROSS REFERENCE
*
*****
```

Errors: none
Warnings: none

Notice that, although the link file specified the address for all segments, many of the segments were not used. The most important information about segments is at the end, where their address and range is given.

Several entry points were described that do not appear in the original C code. The entry for ?C_EXIT is from the CSTARTUP module. The putchar entry is from the library file.

RUNNING THE PROGRAM

To run the program using C-SPY enter the command:

```
csh8 aout -v0 ↵
```

Open the **Memory** window by typing:

```
MEMORY 0 ↵
```

This displays the current contents of memory from address 0, where the data memory variables are located.

Then enter the command:

```
STEP ↵
```

Repeat this until the line reading `do_foreground_process();` is highlighted.

You should see a C-SPY display similar to this:

```

tutor1 #26
int inc_var=1;
char inc_char;
inc_char='b';
}

void main(void)
{
int my_int=0;
call_count=0;
my_char=con_char;
set_local();
while (my_int<100)
{ do_foreground_process();
  my_int++;
}
}

```

```

Terminal I/O
-----
- C-SPY
--> step
--> step
--> step
--> _

```

(c) IAR Systems

Now inspect the value of the variable `call_count` by entering:

```
call_count ↵
```

C-SPY should display 0, since the variable has been initialized but not yet incremented.

Now, enter the command:

```
STEP ↵
```

This displays the current contents of memory from address 0 (where the data memory variables are located). The next step executes the current line and moves to the next line in the loop. Now examine the variable again by entering:

```
call_count ↵
```

C-SPY should display 1, showing that the variable has been incremented by `do_foreground_process()`. The memory contents at address 0001 will be incremented in the **Memory** window.

Enter:

```
LEVEL ↵
```

```
STEP ↵
```

The assembler code for the C program is displayed and the steps are by assembler lines, rather than by C lines. Note that the address of the code is based on the specification in the linker command file.

Enter:

```
LEVEL ↵  
STEP 11 ↵
```

This returns to C level and steps through 11 instructions.

You can modify variables or memory contents while you are debugging. For example, enter:

```
EXPR my_char='c' ↵  
STEP 11 ↵
```

Since `my_char` is not modified within the loop, the subroutine uses the new value.

To quit from C-SPY, enter the command:

```
QUIT ↵
```

MODIFYING THE COMPILE AND LINK OPTIONS

Different compile or link options will produce similar output but with different memory locations. Some options will be explained in the other tutorials, but you may be interested in trying the following examples:

- ◆ Use the **Large** (`-m1`) memory model option instead of the **Small** (`-ms`) option for compiling. See where variables have been placed by examining the map file.
- ◆ Edit the link file to match the change to the large memory model by changing the library loaded to `clh8h1`.

EXTENDING THE PROGRAM

We shall now extend the program to access the serial I/O channel built in to the H8/3003H microprocessor. The resultant program accepts input from serial port number 0 and stores the characters in a buffer. This serial program demonstrates using the `#pragma` directive and header files.

The following is a complete listing of the program. Enter it into a suitable text editor and save it as `tutor2.c`. Alternatively, a copy is provided in the `icch8` subdirectory:


```
/* Enable use of extended keywords */
#pragma language=extended

/* Include sfr definitions for IO registers */
#include <ioh83003.h>

sfr scio_ssr = 0xFFFFFB4; /* Define SFR */
bit RxReady = scio_ssr.6; /* Define bit variable */

/* Mode register bits */
#define CommsMethod (0) /* 8 bit UART mode */
#define ClockSelect (0) /* With internal clock */

/* Main clock divider rate */
#define ClockDivider (32) /* Gives 9600 baud with
                          10MHz processor */

/* Control register bits */
#define ClockRate (0) /* Processor clock */
#define EnableRx (0x10)

#define buffsize 0xC0
char buffer[buffsize];
short buffindex = 0;
short call_count = 0;

/*****
 * Start of Code *
*****/

/*
 * character_ready(void)
 *
 * Return: 0 if no character available
 *         != 0 if character now in data register
 */
short character_ready(void)
{
    return RxReady;
}
```

```
/*
 * read_char(void)
 *
 * Character reader:
 *   poll status register until ready, return data.
 */

char read_character(void)
{
    /* Wait for receive data */
    while (!character_ready())
        ;

    /* Return low 8 bits of receive register */
    return SCIO_RDR;
}

void do_foreground_process(void)
{
    /* Just increment a variable */
    call_count++;
}

void main(void)
{
    /* Initialize comms channel */
    /* Start with mode register */
    SCIO_SMR = CommsMethod + ClockRate;
    SCIO_SCR = ClockSelect + EnableRx;

    /* Now the baud rate */
    SCIO_BRR = ClockDivider;

    /* Now loop forever, taking input when ready */
    while (1)
    {
        if (character_ready())
        {
            buffer[bufferindex++] = read_character();
        }
    }
}
```

```

        /* Full buffer? */
        if (buffindex == buffsize)
            buffindex = 0;          /* Simple processing:
                                   discard data! */
    }
    do_foreground_process();
}
}

```

The first lines of the program are:

```

/* Enable use of extended keywords */
#pragma language=extended

```

By default, extended keywords are not available so you must include this directive before attempting to use any. The `#pragma` directive is described in the chapter *#pragma directive reference*.

The next lines of code are:

```

/* Include sfr definitions for I/O registers */
#include <ioh83003.h>

```

The file `ioh83003.h` includes definitions for all I/O registers for the H8/3003H processor version.

COMPILING AND LINKING THE SERIAL PROGRAM

Compile and link the program with a small memory model and a standard link file as follows:

```

icch8 tutor2 -v0 -ms -r -L -e -q -I\iar\inc ↵
xlink tutor2 -f lnkh8s -r ↵

```

RUNNING THE SERIAL PROGRAM

This program will require a special hardware environment to run properly, but we can examine the code with C-SPY. As before, to run the program, enter:

```

csh8 aout -v0 ↵

```

In C-SPY enter the command:

```

STEP ↵

```

and repeat this until the program reaches the line:

```

if (character_ready ())

```

On a real target with serial input port number 0 connected to a transmitter, a received character would set the 'byte received' flag in bit 0 of the RxReady register and be transferred to the SCIO_RDR register. To simulate the reception of a byte with value 42 first set the 'byte received' flag by entering:

```
RxReady=1 ↵
```

and then set the SCIO_RDR register to 42:

```
SCIO_RDR=42 ↵
```

Now enter:

```
STEP ↵
```

repeatedly until the program has received the byte, and stored it in the buffer; ie until the program has passed the line:

```
buffer[bufferindex++]=read_character();
```

The real serial port would clear its 'byte received' flag when the received byte was read. To simulate this clear the flag directly by entering:

```
RxReady=0 ↵
```

At this point, the received byte should have been transferred into the buffer at the `bufferindex` position, and `bufferindex` should have been advanced by one. To verify this, examine the first byte in the buffer by entering:

```
buffer[0] ↵
```

C-SPY should display 42, the value of the received byte.

Now examine the buffer index by entering:

```
bufferindex ↵
```

C-SPY should display the value 1, showing that the buffer index has advanced from 0.

Finally quit from C-SPY by entering:

```
QUIT ↵
```

ADDING AN INTERRUPT HANDLER

We shall now modify the first tutorial program by adding an interrupt handler. The H8 C Compiler lets you write interrupt handlers directly in C using the `interrupt` keyword. The interrupt we will handle is the serial interrupt.

The following is a complete listing of the interrupt program. The program is provided in the sample tutorials as `tutor3.c`.

```
#pragma language=extended

#include <ioh83003.h>
#include <inh8.h>

/* Channel 0 */
#define MA_SMRO_SCI      0x00 /* Constant for Serial
                               Mode Register */
#define MA_SCRO_SCI     0xF0 /* Constant for Serial
                               Control Register */
#define MA_BRRO_SCI     0x33 /* Constant for Serial
                               Baudrate Register */

unsigned char my_char;
int call_count;

interrupt [SCI_RXIO] void MA_IntHandler_RXIO_SCI(void)
{
    /* Read character */
    my_char = SCIO_RDR;

    SCIO_SSR &= ~0x40;    /* RDRF is cleared */

    return;
}

void do_foreground_process (void)
{
    call_count++;
}

void main (void)
```

```

{
    unsigned long ii;

    /*--- Initialize SCI Channel 0 ---*/
    SCIO_SCR = 0;          /* Clear TE and RE bits */
    SCIO_SMR = MA_SMR0_SCI;
    SCIO_BRR = MA_BRRO_SCI;

    /*--- wait at least one bit time before RE & TE may
    be set ---*/
    for (ii = 0; ii < 100000L; )
    {
        ii++;
    }
    SCIO_SCR = MA_SCR0_SCI & ~0x80;    /* TIE interrupt
                                        disabled */

    /* enable_interrupt */
    set_interrupt_mask (0);
    /* now loop forever, taking input when ready */
    while (1)
    {
        do_foreground_process ();
    }
}

```

The I/O include file must be present to define the H8/300H I/O registers, and the intrinsic include file must be present to define the enable_interrupt function:

```

/* enable use of extended keywords */
#pragma language=extended

/* include sfr definitions for IO registers */
#include <ioh83003.h>
#include <inh8.h>

```

The interrupt function itself is defined by the following lines:

```

interrupt [SCI_RXIO] void MA_IntHandler_RXIO_SCI(void)
{

```

```
/* Read character */
my_char = SCIO_RDR;

SCIO_SSR &= ~0x40;      /* RDRF is cleared */

return;

}
```

The action of this program is to output a character to port 1, making it easy to identify the event.

The interrupt keyword is described in the chapter *Extended keyword reference*.

COMPILING AND LINKING THE PROGRAM

Compile and link the program as before:

```
icch8 tutor3 -v0 -ms -r -L -e -q -I\iar\inc ↵
xlink tutor3 -f lnkh8hs -r ↵
```

VIEWING THE INTERRUPT PROGRAM

Note that this program needs hardware to run properly, but we can examine the code with C-SPY. As before, to run the program, enter:

```
csh8 -v0 aout ↵
```

followed by:

```
STEP ↵
```

```
LEVEL ↵
```

C-SPY does not simulate interrupts, but you can use the LEVEL command to examine the assembly code produced. Alternatively, examine the list file output on a printed copy.

C COMPILER OPTIONS

SUMMARY

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

The options are divided into the following sections:

Code generation	#undef
Debug	Include
#define	Target
List	Miscellaneous

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

SETTING C COMPILER OPTIONS

To set C compiler options you include them on the command line after the `icch8` command, either before or after the source filename. For example, when compiling the source `prog`, to generate a listing to the default listing filename (`prog.lst`):

```
icch8 prog -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`:

```
icch8 prog -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`:

```
icch8 prog -Llist ↵
```

Generally, the order of options on the command line, both relative to each other and to the source filename, is not significant. The exception is that the order in which two or more `-I` options are used is significant.

Options can also be specified in the `QCCH8` environment variable. The compiler 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 compilation.

OPTIONS SUMMARY

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

<i>Option</i>	<i>Description</i>	<i>Section</i>
- <i>Aprefix</i>	Assembly output to prefixed filename.	List
-a <i>filename</i>	Assembly output to named file.	List
-b	Make object a library module.	Miscellaneous
-C	Nested comments.	Code generation
-c	Char is signed char.	Code generation
-D <i>symb</i> [<i>xx</i>]	Defined symbols.	#define
-d	Static allocation of locals.	Code generation
-e	Language extensions.	Code generation
-F	Form-feed after function.	List
-f <i>filename</i>	Extend the command line.	Miscellaneous
-G	Open standard input as source.	Miscellaneous
-g	Global strict type check.	Code generation
-gA	Flag old-style functions.	Code generation
-g0	No type info. in obj. code.	Code generation
-H <i>name</i>	Set object module name.	Miscellaneous
-I <i>prefix</i>	Include paths.	Include
-i	Add #include file lines.	List
-K	// comments.	Code generation
-L[<i>prefix</i>]	List to prefixed source name.	List
-l <i>filename</i>	List to named file.	List
-m[<i>s</i>]	Memory model.	Target
-N <i>prefix</i>	Preprocessor to prefixed filename.	List
-n <i>filename</i>	Preprocessor to named file.	List

<i>Option</i>	<i>Description</i>	<i>Section</i>
-O <i>prefix</i>	Set object filename prefix.	Miscellaneous
-o <i>filename</i>	Set object filename.	Miscellaneous
-P	Generate PROMable code.	Miscellaneous
-p <i>nn</i>	Lines/page.	List
-q	Insert mnemonics.	List
-R <i>name</i>	Set code segment name.	Miscellaneous
-r[012][i][n]	Generate debug information.	Debug
-S	Set silent operation.	Miscellaneous
-s[0-9]	Optimize for speed.	Code generation
-T	Active lines only.	List
-t <i>n</i>	Tab spacing.	List
-Usymb	Undefine symbol.	#undef
-ui	Run-time library function calls.	Code generation
-um <i>n</i>	Interrupt disable max time.	Code generation
-v <i>n</i>	Chip option.	Target
-W <i>nn</i>	Stack optimization limit.	Code generation
-w	Disable warnings.	Code generation
-X	List C declarations.	List
-x[DFT2]	Cross reference.	List
-y	Writable strings.	Code generation
-z[0-9]	Optimize for size.	Code generation
-2	64-bit floating point.	Target

C COMPILER OPTIONS REFERENCE

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

CODE GENERATION

The code generation options determine the interpretation of the source program and the generation of object code.

-C	Nested comments.
-c	Char is signed char.
-d	Static allocation of locals.
-e	Language extensions.
-g	Global strict type check.
-gA	Flag old-style functions.
-g0	No type information in object code.
-K	// comments.
-s[0-9]	Optimize for speed.
-ui	Run-time library function calls.
-umn	Interrupt disable max time.
-Wnn	Stack optimization limit.
-w	Disable warnings.
-y	Writable strings.
-z[0-9]	Optimize for size.

NESTED COMMENTS (-C)

Syntax: -C

Enables nested comments.

Normally, the compiler treats nested comments as a fault and issues a warning when it encounters one, resulting for example from a failure to close a comment. If you want to use nested comments, for example to comment-out sections of code that include comments, use the **Nested comments** (-C) option to disable this warning.

CHAR IS SIGNED CHAR (-c)

Syntax: -c

Makes the char type equivalent to signed char.

Normally, the compiler interprets the char type as unsigned char. To make the compiler interpret the char type as signed char instead, for example for compatibility with a different compiler, use this option.

Note: the run-time library is compiled without the **Char is signed char** (-c) option, so if you use this option for your program and enable type checking with the **Global strict type check** (-g) or **Generate debug information** (-r) options, you may get type mismatch warnings from the linker.

STATIC ALLOCATION OF LOCAL VARIABLES (-d)

Syntax: -d

Causes the compiler to allocate static memory for local variables.

Normally the compiler allocates local variables on the stack.

LANGUAGE EXTENSIONS (-e)

Syntax: -e

Enables target dependent extensions to the C language.

Normally, language extensions are disabled to preserve compatibility. If you are using language extensions in the source, you must enable them by including this option.

For details of language extensions, see the chapter *Language extensions*.

GLOBAL STRICT TYPE CHECK (-g)

Syntax: -g[A][0]

Enable checking of type information throughout the source.

There is a class of conditions in the source that indicate possible programming faults but which for compatibility the compiler and linker normally ignore. To cause the compiler and linker to issue a warning each time they encounter such a condition, use the **Global strict type check** (-g) option.

FLAG OLD-STYLE FUNCTIONS (-gA)

Syntax: -gA

Normally, the **Global strict type check** (-g) option does not warn of old-style K&R functions. To enable such warnings, use the **Flag old-style functions** (-gA) option.

NO TYPE INFORMATION IN OBJECT CODE (-g0)

Syntax: -g0

Normally, the **Global strict type check** (-g) option includes type information in the object module, increasing its size and link time, allowing the linker to issue type check warnings. To exclude this information, avoiding this increase in size and link time but inhibiting linker type check warnings, use the **No type information in object code** (-g0) option.

When linking multiple modules, note that objects in a module compiled without type information, that is without any -g option or with a -g option with 0 modifier, are considered typeless. Hence there will never be any warning of a type mismatch from a declaration from a module compiled without type information, even if the module with a corresponding declaration has been compiled with type information.

The conditions checked by the **Global strict type check** (-g) option are:

- ◆ Calls to undeclared functions.
- ◆ Undeclared K&R formal parameters.
- ◆ Missing return values in non-void functions.

- ◆ Unreferenced local or formal parameters.
- ◆ Unreferenced goto labels.
- ◆ Unreachable code.
- ◆ Unmatching or varying parameters to K&R functions.
- ◆ #undef on unknown symbols.
- ◆ Valid but ambiguous initializers.
- ◆ Constant array indexing out of range.

Examples

The following examples illustrate each of these types of error.

Calls to undeclared functions

Program:

```
void my_fun(void) { }
int main(void)
{
    my_func();    /* mis-spelt my_fun gives undeclared
                  function warning */
    return 0;
}
```

Error:

```
my_func();        /* mis-spelt my_fun gives undeclared
                  function warning */
-----^
"undecfn.c",5 Warning[23]: Undeclared function
'my_func'; assumed "extern" "int"
```

Undeclared K&R formal parameters

Program:

```
int my_fun(parameter) /* type of parameter not declared
                      */
{
    return parameter+1;
}
```

Error:

```
int my_fun(parameter) /* type of parameter not declared
                      */
```



```
-----^
"undecfp.c",1 Warning[9]: Undeclared function parameter
'parameter'; assumed "int"
```

Missing return values in non-void functions

Program:

```
int my_fun(void)
{
    /* ... function body ... */
}
```

Error:

```
}
^
"noreturn.c",4 Warning[22]: Non-void function: explicit
"return" <expression>; expected
```

Unreferenced local or formal parameters

Program:

```
void my_fun(int parameter)      /* unreferenced formal
                                parameter */
{
    int localvar;              /* unreferenced local
                                variable */
    /* exit without reference to either variable */
}
```

Error:

```
}
^
"unrefpar.c",6 Warning[33]: Local or formal 'localvar'
was never referenced
"unrefpar.c",6 Warning[33]: Local or formal 'parameter'
was never referenced
```

Unreferenced goto labels

Program:

```
int main(void)
{
    /* ... function body ... */
    exit:                          /* unreferenced label */
}
```

```
    return 0;
}
```

Error:

```
}
^
```

"unreflab.c",7 Warning[13]: Unreferenced label 'exit'

Unreachable code

Program:

```
#include <stdio.h>
int main(void)
{
    goto exit;
    puts("This code is unreachable");
exit:
    return 0;
}
```

Error:

```
    puts("This code is unreachable");
-----^
```

"unreach.c",7 Warning[20]: Unreachable statement(s)

Unmatching or varying parameters to K&R functions

Program:

```
int my_fun(len,str)
int len;
char *str;
{
    str[0]='a' ;
    return len;
}
char buffer[99] ;
int main(void)
{
    my_fun(buffer,99) ; /* wrong order of parameters */
    my_fun(99) ; /* missing parameter */
    return 0 ;
}
```

Error:

```
my_fun(buffer,99) ;      /* wrong order of parameters */
-----^
"varyparm.c",14 Warning[26]: Inconsistent use of K&R
function - changing type of parameter
my_fun(buffer,99) ;      /* wrong order of parameters */
-----^
"varyparm.c",14 Warning[26]: Inconsistent use of K&R
function - changing type of parameter
my_fun(99) ;            /* missing parameter */
-----^
"varyparm.c",15 Warning[25]: Inconsistent use of K&R
function - varying number of parameters
```

#undef on unknown symbols

Program:

```
#define my_macro 99
/* Misspelt name gives a warning that the symbol is
unknown */
#undef my_macor
int main(void)
{
    return 0;
}
```

Error:

```
#undef my_macor
-----^
"hundef.c",4 Warning[2]: Macro 'my_macor' is already
#undef
```

Valid but ambiguous initializers

Program:

```
typedef struct t1 {int f1; int f2;} type1;
typedef struct t2 {int f3; type1 f4; type1 f5;} type2;
typedef struct t3 {int f6; type2 f7; int f8;} type3;
type3 example = {99, {42,1,2}, 37} ;
```

Error:

```
type3 example = {99, {42,1,2}, 37} ;
-----^
```

"ambigini.c",4 Warning[12]: Incompletely bracketed initializer

Constant array indexing out of range

Program:

```
char buffer[99] ;
int main(void)
{
    \buffer[500] = 'a' ; /* Constant index out of range */
    return 0;
}
```

Error:

```
\buffer[500] = 'a' ;    /* Constant index out of range */
-----^
```

"arrindex.c",5 Warning[28]: Constant [index] outside array bounds

// COMMENTS (-K)

Syntax: -K

Enables comments in C++ style, that is, comments introduced by // and extending to the end of the line.

Normally for compatibility the compiler does not accept C++ style comments. If your source includes C++ style comments, you must use the // **comments** (-K) option for them to be accepted.

OPTIMIZE FOR SPEED (-s)

Syntax: -s[0-9]

Causes the compiler to optimize the code for maximum execution speed.

Normally the compiler optimizes for maximum speed at level 3 (see below). You can change the level of optimization using the -s option as follows:

<i>Modifier</i>	<i>Level</i>
0	No optimization.
1-3	Fully debuggable.

<i>Modifier</i>	<i>Level</i>
4-6	Some constructs not debuggable.
7-9	Full optimization.

RUN-TIME LIBRARY FUNCTION CALLS (-ui)

Syntax: -ui

Changes the default setting for run-time library calls, as follows:

<i>Memory model</i>	<i>Default</i>	<i>-ui</i>
Small	tiny_func	near_func
Large	far_func	tiny_func

INTERRUPT DISABLE MAX TIME (-um)

Syntax: -umn

Specifies the maximum time, in cycles, for which interrupts can be disabled.

STACK OPTIMIZATION LIMIT (-W)

Syntax: -Wnn

The number of clean-ups of the stack is reduced by specifying stack optimization with the -W option. For example, by specifying -W50, the compiler is instructed to allow 50 bytes of garbage on the stack before triggering stack clean-up. The default setting is -W0, ie no stack optimization.

DISABLE WARNINGS (-w)

Syntax: -w

Disables compiler warning messages.

Normally, the compiler issues standard warning messages, and any additional warning messages enabled with the **Global strict type check** (-g) option. To disable all warning messages, you use the **Disable warnings** (-w) option.

WRITABLE STRINGS (-y)**Syntax:** -y

Causes the compiler to compile string literals as writable variables.

Normally, string literals are compiled as read-only. If you want to be able to write to string literals, you use the **Writable strings** (-y) option, causing strings to be compiled as writable variables.

Note that arrays initialized with strings (ie `char c[] = "string"`) are always compiled as initialized variables, and are not affected by the **Writable strings** (-y) option.

OPTIMIZE FOR SIZE (-z)**Syntax:** -z[0-9]

Causes the compiler to optimize the code for minimum size.

Normally, the compiler optimizes for minimum size at level 3 (see below). You can change the level of optimization as follows:

<i>Modifier</i>	<i>Level</i>
0	No optimization.
1-3	Fully debuggable.
4-6	Some constructs not debuggable.
7-9	Full optimization.

DEBUG

The **Debug** options determine the level of debugging information included in the object code.

-r[012][i][n] Generate debug information.

GENERATE DEBUG INFORMATION (-r)**Syntax:** -r[012][i][n]

Causes the compiler to include additional information required by C-SPY and other symbolic debuggers in the object modules.

Normally the compiler does not include debugging information, for code efficiency. To make code debuggable with C-SPY, you simply include the option with no modifiers.

To make code debuggable with other debuggers, you select one or more options, as follows:

<i>Option</i>	<i>Command line</i>
Add <code>#include</code> file information.	<code>i</code>
Suppress source in object code.	<code>n</code>
Code added to statements.	<code>0, 1, 2</code>

Normally the **Generate debug information** (`-r`) option does not include `#include` file debugging information, because this is usually of little interest, and most debuggers other than C-SPY do not support debugging inside `#include` files well. If you want to debug inside `#include` files, for example if the `#include` files contain function definitions rather than the more usual function declarations, you use the `i` modifier. A side effect is that source line records contain the global (= total) line count which can affect source line displays in some debuggers other than C-SPY.

The **Generate debug information** (`-r`) option usually includes C source lines in the object file, so they can be displayed during debugging. If you want to suppress this to reduce the size of the object file, you use the `n` modifier.

For most other debuggers that do not include specific information on how to use IAR Systems C Compilers, you should use the `-rn` option.

#define

The **#define** option allows you to define symbols for use by the C compiler.

`-D` Defined symbols.

DEFINED SYMBOLS (-D)

Syntax: `-D $symbol$ [xx]`

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

Defined symbols (-D) has the same effect as a `#define` statement at the top of the source file.

`-Dsymb` is equivalent to `#define symb`

The **Defined symbols** (-D) option is useful for specifying a value or choice that would otherwise be specified in the source file more conveniently on the command line. For example, you could arrange your source to produce either the test or production version of your program depending 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: `icch8 prog ↵`

test version: `icch8 prog -Dtestver ↵`

LIST

The **List** options determine whether a listing is produced, and the information included in the listing.

<code>-A<i>prefix</i></code>	Assembly output to prefixed filename.
<code>-a <i>filename</i></code>	Assembly output to named file.
<code>-F</code>	Form-feed after function.
<code>-i</code>	Add <code>#include</code> file lines.
<code>-L[<i>prefix</i>]</code>	List to prefixed source name.
<code>-l <i>filename</i></code>	List to named file.
<code>-N<i>prefix</i></code>	Preprocessor to prefixed filename.
<code>-n <i>filename</i></code>	Preprocessor to named file.
<code>-pnn</code>	Lines/page.
<code>-q</code>	Insert mnemonics.
<code>-T</code>	Active lines only.

-tn	Tab spacing.
-X	List C declarations.
-x[DFT2]	Cross reference.

ASSEMBLY OUTPUT TO PREFIXED FILENAME (-A)

Syntax: -A*prefix*

Generates assembler source to *prefix source.s37*.

By default the compiler does not generate an assembler source. To send assembler source to the file with the same name as the source leafname but with the extension *.s37*, use -A without an argument. For example:

```
icch8 prog -A ↵
```

generates an assembly source to the file *prog.s37*.

To send assembler source to the same filename but in a different directory, use the -A option with the directory as the argument. For example:

```
icch8 prog -Aasm\ ↵
```

generates an assembly source in the file *asm\prog.s37*.

The assembler source may be assembled by the H8 Assembler.

If the -l or -L option is also used, the C source lines are included in the assembly source file as comments.

The -A option may not be used at the same time as the -a option.

ASSEMBLY OUTPUT TO NAMED FILE (-a)

Syntax: -a *filename*

Generates assembler source to *filename.s37*.

By default the compiler does not generate an assembler source. This option generates an assembler source to the named file.

The filename consists of a leafname optionally preceded by a pathname and optionally followed by an extension. If no extension is given, the target-specific assembler source extension is used.

The assembler source may be assembled by the H8 Assembler.

If the `-l` or `-L` option is also used, the C source lines are included in the assembly source file as comments.

This option may not be used at the same time as `-A`.

FORM-FEED AFTER FUNCTION (-F)

Syntax: `-F`

Generates a form-feed after each listed function in the assembly listing.

Normally, the listing simply starts each function on the next line. To cause each function to appear at the top of a new page, you would include this option.

Form-feeds are never generated for functions that are not listed, for example, as in `#include` files.

ADD #INCLUDE FILE LINES (-i)

Syntax: `-i`

Causes the listing to include `#include` files.

Normally the listing does not include `#include` files, since they usually contain only header information that would waste space in the listing. To include `#include` files, for example because they include function definitions or preprocessed lines, you include the **Add #include file lines (-i)** option.

LIST TO PREFIXED SOURCE NAME (-L)

Syntax: `-L[prefix]`

Generate a listing to the file with the same name as the source but with extension `.lst`, prefixed by the argument if any.

Normally, the compiler does not generate a listing. To simply generate a listing, you use the `-L` option without a prefix. For example, to generate a listing in the file `prog.lst`, you use:

```
icch8 prog -L ↵
```

To generate a listing to a different directory, you use the `-L` option followed by the directory name. For example, to generate a listing on the corresponding filename in the directory `\list`:

```
icch8 prog -Llist\ ↵
```

This sends the file 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`

Generates a listing to the named file with the default extension `.lst`.

Normally, the compiler does not generate a listing. To generate a listing to a named file, you use the `-l` option. For example, to generate a listing to the file `list.lst`, use:

```
icch8 prog -l list
```

More often you do not need to specify a particular filename, in which case you can use the `-L` option instead.

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

PREPROCESSOR TO PREFIXED FILENAME (-N)

Syntax: `-Nprefix`

Generates preprocessor output to `prefix source.i`.

By default the compiler does not generate preprocessor output. To send preprocessor output to the file with the same name as the source leafname but with the extension `.i`, use the `-N` without an argument. For example:

```
icch8 prog -N
```

generates preprocessor output to the file `prog.i`.

To send preprocessor output to the same filename but in a different directory, use the `-N` option with the directory as the argument. For example:

```
icch8 prog -Npreproc
```

generates an assembly source in the file `preproc\prog.i`.

The `-N` option may not be used at the same time as the `-n` option.

PREPROCESSOR TO NAMED FILE (-n)

Syntax: -n *filename*

Generates preprocessor output to *filename.i*.

By default the compiler does not generate preprocessor output. This option generates preprocessor output to the named file.

The filename consists of a leafname optionally preceded by a pathname and optionally followed by an extension. If no extension is given, the extension *.i* is used.

This option may not be used at the same time as -N.

LINES/PAGE (-p)

Syntax: -p*nn*

Causes the listing to be formatted into pages, and specifies the number of lines per page in the range 10 to 150.

Normally, the listing is not formatted into pages. To format it into pages with a form feed at every page, you use the **Lines/page** (-p) option. For example, for a printer with 50 lines per page:

```
icch8 prog -p50 ↵
```

INSERT MNEMONICS (-q)

Syntax: -q

Includes generated assembly lines in the listing.

Normally, the compiler does not include the generated assembly lines in the listing. If you want these to be included, for example to be able to check the efficiency of code generated by a particular statement, you use the **Insert mnemonics** (-q) option.

Note that this option is only available if a listing is specified.

See also options -a, -A, -l, and -L.

ACTIVE LINES ONLY (-T)**Syntax:** -T

Causes the compiler to list only active source lines.

Normally the compiler lists all source lines. To save listing space by eliminating inactive lines, such as those in false `#if` structures, you use the **Active lines only** (-T) option.

TAB SPACING (-t)**Syntax:** -tn

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

Normally, the listing is formatted with a tab spacing of 8 characters. If you want a different tab spacing, you set it with the **Tab spacing** (-t) option.

LIST C DECLARATIONS (-X)**Syntax:** -X

Displays an English description of each C declaration in the file.

To obtain English descriptions of the C declarations, for example to aid the investigation of error messages, you use the **List C declarations** (-X) option.

For example, the declaration:

```
void (* signal(int __sig, void (* func) ())) (int);
```

gives the description:

```
Identifier: signal
storage class: extern
[func_attr:0220] prototyped near_func function returning
[attribute:0120] near - near_func code pointer to
[func_attr:0220] prototyped near_func function
returning
[attribute:0120] near - void
and having following parameter(s):
storage class: auto
[attribute:0120] near - int
```

```
and having following parameter(s):
storage class: auto
[attribute:0120] near - int
storage class: auto
[attribute:0120] near - near_func code pointer to
[func_attr:0220] near_func function returning
[attribute:0120] near - void
```

CROSS REFERENCE (-x)

Syntax: -x[DFT2]

Includes a cross-reference list in the listing.

Normally, the compiler does not include global symbols in the listing. To include at the end of the listing a list of all variable objects, and all functions, `#define` statements, enum statements, and `typedef` statements that are referenced, you use the **Cross reference** (-x) option with no modifiers.

When you select **Cross reference** the following options become available:

<i>Command line</i>	<i>Option</i>
D	Show unreferenced <code>#defines</code> .
T	Show unreferenced <code>typedefs</code> and enum constants.
F	Show unreferenced functions.
2	Dual line spacing.

#undef

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

`-U $symb$` Undefine symbol.

UNDEFINE SYMBOL (-U)

Syntax: -U $symb$

Removes the definition of the named symbol.

Normally, the compiler provides various pre-defined symbols. If you want to remove one of these, for example to avoid a conflict with a

symbol of your own with the same name, you use the **Undefine symbol** (-U) option.

For a list of the predefined symbols, see the chapter *Predefined symbols reference*.

For example, to remove the symbol `__VER__`, use:

```
icch8 prog -U__VER__ ↵
```

INCLUDE

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

`-Iprefix` Include paths.

INCLUDE PATHS (-I)

Syntax: `-Iprefix`

Adds a prefix to the list of `#include` file prefixes.

Normally, the compiler searches for include files only in the source directory (if the filename is enclosed in quotes as opposed to angle brackets), the `C_INCLUDE` paths, and finally the current directory. If you have placed `#include` files in some other directory, you must use the **Include paths** (-I) option to inform the compiler of that directory.

For example:

```
icch8 prog -I\mylib\ ↵
```

Note that the compiler simply adds the -I prefix onto the start of the include filename, so it is important to include the final backslash if necessary.

There is no limit to the number of -I options allowed on a single command line. When many -I options are used, to avoid the command line exceeding the operating system's limit, you would use a command file; see the -f option.

Note: the full description of the compiler's `#include` file search procedure is as follows:

When the compiler encounters an include file name in angle brackets such as:

```
#include <stdio.h>
```

it performs the following search sequence:

- ◆ The filename prefixed by each successive -I prefix.
- ◆ The filename prefixed by each successive path in the C_INCLUDE environment variable if any.
- ◆ The filename alone.

When the compiler encounters an include file name in double quotes such as:

```
#include "vars.h"
```

it searches the filename prefixed by the source file path, and then performs the sequence as for angle-bracketed filenames.

TARGET

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

- m[sl] Memory model.
- vn Chip option.
- 2 64-bit floating point.

MEMORY MODEL (-m)

Syntax: -m[sl]

Selects the memory model for which the code is to be generated, as follows:

<i>Option</i>	<i>Command line</i>
Small (default)	s
Large	l

The memory model determines the maximum size of code and maximum size of data normally available.

Normally, the compiler generates code for the small memory model. If you want code for the large memory model, you use the -ml option.

The -ms option is provided for consistency, and though not necessary, you may use it to make clear you are using the small memory model.

CHIP OPTION (-v)**Syntax:** - *vn*

Selects the processor version as follows:

<i>Option</i>	<i>Description</i>
-v0	H8/300H (default)
-v1	H8S/2200
-v2	H8S/2600

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

Note that changing the processor group causes different code to be generated. For information about the addressing supported by each processor group see *Memory model*, page 64.

64-BIT FLOATING POINT (-2)**Syntax:** - 2

Selects 64-bit IEEE floating point format for doubles and long doubles.

Normally the compiler uses 32-bit precision for doubles and long doubles.

MISCELLANEOUS

The following additional option is available from the command line.

-b	Make object a library module.
-f <i>filename</i>	Extend the command line.
-G	Open standard input as source.
-H <i>name</i>	Set object module name.
-O <i>prefix</i>	Set object filename prefix.
-o <i>filename</i>	Set object filename.
-P	Generate PROMable code.
-R <i>name</i>	Set code segment name.
-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.

The compiler normally produces a program module ready for linking with XLINK. If instead you want a library module for inclusion in a library with XLIB, you use the -b option.

EXTEND THE COMMAND LINE (-f)

Syntax: -f *filename*

Reads command line options from the named file, with the default extension .xcl.

Normally, the compiler accepts command parameters only from the command line itself and the QCCH8 environment variable. To make long command lines more manageable, and to avoid any operating system command line length limit, you use the -f option to specify a command file, from which the compiler reads command line items as if they had been entered at the position of the option.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines since the newline character acts just as a space or tab character.

For example, you could replace the command line:

```
icch8 prog -r -L -Dtestver "-Dusername=John Smith"  
-Duserid=463760 ↵
```

with

```
icch8 prog -r -L -Dtestver -fuserinfo ↵
```

and the file userinfo.xcl containing:

```
"-Dusername=John Smith"  
-Duserid=463760
```

OPEN STANDARD INPUT AS SOURCE (-G)**Syntax:** -G

Opens the standard input as source, instead of reading source from a file.

Normally, the compiler reads source from the file named on the command line. If you wish it to read source instead from the standard input (normally the keyboard), you use the -G option and omit the source filename.

The source filename is set to `stdin.c`.

SET OBJECT MODULE NAME (-H)**Syntax:** -H*name*

Normally, the internal name of the object module is the name of the source file, without directory name or extension. To set the object module name explicitly, you use the -H option, for example:

```
icch8 prog -Hmain ↵
```

This is particularly useful when several modules have the same filename, since normally the resulting duplicate module name would cause a linker error. An example is when the source file is a temporary file generated by a preprocessor. The following (in which %1 is an operating system variable containing the name of the source file) will give duplicate name errors from the linker:

```
preproc %1.c temp.c      ; preprocess source, generating
                          temp.c
icch8 temp.c             ; module name is always 'temp'
```

To avoid this, use -H to retain the original name:

```
preproc %1.c temp.c      ; preprocess source, generating
                          temp.c
icch8 temp.c -H%1        ; use original source name as
                          module name
```

SET OBJECT FILENAME PREFIX (-O)**Syntax:** *-Oprefix*

Sets the prefix to be used on the filename of the object.

Normally (and unless the `-o` option is used) the object is stored with the filename corresponding to the source filename, but with the extension `.r37`. To store the object in a different directory, you use the `-O` option.

For example, to store the object in the `\obj` directory, use:

```
icch8 prog -O\obj\ ↵
```

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

SET OBJECT FILENAME (-o)**Syntax:** *-o filename*

Set the filename in which the object module will be stored. The filename consists of an optional pathname, obligatory leafname, and optional extension (default `.r37`).

Normally the compiler stores the object code in a file whose name is:

- ◆ The prefix specified by `-o`, plus
- ◆ The leafname of the source, plus
- ◆ The extension `.r37`.

To store the object in a different filename, you use the `-o` option. For example, to store it in the file `obj.r32`, you would use:

```
icch8 prog -o prog ↵
```

If instead you want to store the object with the corresponding filename but in a different directory, use the `-O` option.

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

GENERATE PROMABLE CODE (-P)**Syntax:** -P

Causes the compiler to generate code suitable for running in read-only memory (PROM).

This option is included for compatibility with other IAR compilers, but in the H8 C Compiler is always active.

SET CODE SEGMENT NAME (-R)**Syntax:** -R*name*

Sets the name of the code segment.

Normally, the compiler places executable code in the segment named CODE which, by default, the linker places at a variable address. If you want to be able to specify an explicit address for the code, you use the -R option to specify a special code segment name which you can then assign to a fixed address in the linker command file.

SET SILENT OPERATION (-S)**Syntax:** -S

Causes the compiler to operate without sending unnecessary messages to standard output (normally the screen).

Normally the compiler issues introductory messages and a final statistics report. To inhibit this output, you use the -S option. This does not affect the display of error and warning messages.

CONFIGURATION

This chapter describes how to configure the C compiler for different requirements.

INTRODUCTION

Systems based on the H8 microprocessor can vary considerably in their use of ROM and RAM, and in their stack requirements. They also differ in their need for libraries. The memory model and link options specify:

- ◆ The ROM areas: used for functions, constants, and initial values.
- ◆ The RAM areas: used for stack and variables.

Each feature of the environment or usage is handled by one or more configurable elements of the compiler packages, as follows:

<i>Feature</i>	<i>Configurable element</i>	<i>See page</i>
Processor group	Compiler option, XLINK command file (including run-time library).	62
Memory model	Compiler option, XLINK option (including run-time library).	64
Floating-point precision	Compiler option, XLINK command file.	64
putchar and getchar functions	Run-time library module.	66
printf/scanf facilities	XLINK command file.	67, 68
Heap size	Heap library module.	69
Hardware/memory initialization	__low_level_init module.	69

The following sections describe each of the above features. Note that many of the configuration procedures involve editing the standard files, and you may want to make copies of the originals before beginning.

PROCESSOR GROUP

The H8 Series of microprocessors has many variants, which the H8 C Compiler divides into three groups.

SPECIFYING THE PROCESSOR GROUP

Your program may only use one processor group at a time, and the same processor group must be used by all user modules and all library modules.

To specify the processor group to the compiler when a user module is compiled, you use one of the following target options:

<i>Option</i>	<i>Description</i>
-v0	H8/300H (default)
-v1	H8S/2200
-v2	H8S/2600

For example, to compile myprog for use on the H8S/2200 use the command:

```
icch8 myprog -v1 
```

XLINK COMMAND FILE

To create an XLINK command file for a particular project you should first copy the appropriate supplied template from `c:\iar\icch8`. The supplied templates, covering each available memory model of each chip option, are as follows.

<i>Options</i>	<i>H8/300H (-v0)</i>	<i>H8S (-v1 or -v2)</i>
Small memory (-ms)	lnkh8hs.xcl	lnkh8ss.xcl
Large memory (-ml)	lnkh8hl.xcl	lnkh8sl.xcl

You should then modify this file, as described within the file, to specify the details of the target system's memory map.

RUN-TIME LIBRARY

Each template file refers to its appropriate library modules so you will not normally need to specify the library module itself.

The following library modules are supplied:

<i>Options</i>	<i>H8/300H (-v0)</i>	<i>H8S (-v1 or -v2)</i>
Small memory, 32-bit doubles (-ms)	clh8hs.r37	clh8ss.r37
Small memory, 64-bit doubles (-ms, -2)	clh8hsd.r37	clh8ssd.r37
Large memory, 32-bit doubles (-m1)	clh8h1.r37	clh8s1.r37
Large memory, 64-bit doubles (-m1, -2)	clh8h1d.r37	clh8s1d.r37

Predefined special function registers (SFRs) and interrupt routines are given in the following header files:

<i>Description</i>	<i>H8/300H and H8S</i>
Source header for intrinsic functions	inh8.h
Source header for use by printf	iccbut1.h
Source header for internal library definitions	iccext.h
Source header for I/O addresses and interrupt vectors	ioh8xxx.h

These files are provided in the icch8 subdirectory.

MEMORY MODEL

The H8 C Compiler supports the following memory models:

<i>Processor mode</i>	<i>Memory model</i>	<i>Program memory</i>	<i>Data memory</i>	<i>Default func call</i>	<i>Default data pointer</i>
Normal	Small (-ms)	< 64 Kbytes	< 64 Kbytes	near_func	near
Advanced	Large (-ml)	< 16 Mbytes	< 4 Gbytes	far_func	far

The default is the small memory model.

MEMORY LOCATION

You need to specify to XLINK your hardware environment's address ranges for ROM and RAM. You would normally do this in your copy of the XLINK command file template.

For details of specifying the memory address ranges, see the contents of the XLINK command file template and the XLINK section of the *H8 Assembler, Linker, and Librarian Programming Guide*.

NON-VOLATILE RAM

The compiler supports the declaration of variables that are to reside in non-volatile RAM through the `no_init` type modifier and the memory `#pragma`. The compiler places such variables in the separate segment `NO_INIT`, which you should assign to the address range of the non-volatile RAM of the hardware environment. The run-time system does not initialize these variables.

To assign the `NO_INIT` segment to the address of the non-volatile RAM, you need to modify the XLINK command file. For details of assigning a segment to a given address, see the XLINK section of the *H8 Assembler, Linker, and Librarian Programming Guide*.

FLOATING-POINT PRECISION

By default, floating-point numbers of type `double` are represented in IEEE 4-byte format, equivalent to `float`. You can specify the `-2` option to make `double` use IEEE 8-byte format.

All modules in a program must use the same precision of `double`. Notably, this includes the run-time library modules; see *Run-time library*, page 63. For details of the representation of floating-point numbers, see *Data representation*, page 73.

STACK SIZE

The compiler uses a stack for a variety of user program operations, and the required stack size depends heavily on the details of these operations. If the given stack size is too small, the stack will normally be allowed to overwrite variable storage resulting in likely program failure. If the given stack size is too large, RAM will be wasted.

ESTIMATING THE REQUIRED STACK SIZE

The stack is used for the following:

- ◆ Storing local variables and parameters.
- ◆ Storing temporary results in expressions.
- ◆ Storing temporary values in run-time library routines.
- ◆ Saving the return address of function calls.
- ◆ Saving the processor state during interrupts.

The total required stack size is the worst case total of the required sizes for each of the above.

CONTROLLING STACK USAGE

The amount of stack that is used for temporary results and for transferring parameters can be controlled by the stack optimize option `-W`. It sets a limit for the amount of garbage that is allowed on the stack. The space that is allocated for parameters, for instance, becomes garbage when returning from a function. If the stack space is sufficient, a sensible setting of the `-W` option will reduce the number of stack clean-ups required, eg after function calls. By doing so, the produced code gets both faster and smaller.

If no stack optimization is specified, the default setting `-W0` is used, which means that no stack clean-ups are postponed.

CHANGING THE STACK SIZE

The default stack size is set to 512 (200h) bytes in the linker command files, with the expression `CSTACK+200` in the linker command:

```
-Z(DATA)CSTACK+200
```

To change the stack size edit the linker command file and replace 200 by the size of the stack you want to use.

INPUT AND OUTPUT

PUTCHAR AND GETCHAR

The functions `putchar` and `getchar` are the fundamental functions through which C performs all character-based I/O. For any character-based I/O to be available, you must provide definitions for these two functions using whatever facilities the hardware environment provides.

The starting-point for creating new I/O routines is the files `c:\iar\icch8\putchar.c` and `c:\iar\icch8\getchar.c`.

Customizing putchar

The procedure for creating a customized version of `putchar` is as follows:

- ◆ Make the required additions to the source `putchar.c`, and save it back under the same name (or create your own routine using `putchar.c` as a model). The code below uses memory-mapped I/O to write to an LCD display.

```
#include <stdio.h>
int putchar(int outchar)
{
    unsigned char *LCD_IO;
    LCD_IO= (unsigned char *) 0x8000;
    * LCD_IO=outchar;
    return(outchar);
}
```

- ◆ Compile the modified `putchar` using the appropriate processor option. For example, if your program uses the small memory model and the H8/300H, compile `putchar.c` with the command:

```
icch8 putchar -v0 -b ↵
```

This will create an optimized replacement object module file named `putchar.r37`.

- ◆ Add the new `putchar` module to the appropriate run-time library module, replacing the original. For example, to add the new `putchar` module to the standard small-memory-model library, use the command:

```
xlib ↵
def-cpu h8 ↵
rep-mod putchar clh8hs ↵
exit ↵
```

The library module `clh8hs` will now have the modified `putchar` instead of the original. (Be sure to save your original `clh8hs.r37` file before you overwrite the `putchar` module.)

Note that XLINK allows you to test the modified module before installing it in the library by using the `-A` option. Place the following lines into your `.xcl` link file:

```
-A putchar
clh8hs
```

This causes your version of `putchar.r37` to load instead of the one in the `clh8hs` library. See the *H8 Assembler, Linker, and Librarian Programming Guide*. Note that `putchar` serves as the low-level part of the `printf` function.

Customizing getchar

The low-level I/O function `getchar` is supplied as two C files, `getchar.c` and `llget.c`.

PRINTF AND SPRINTF

The `printf` and `sprintf` functions use a common formatter called `_formatted_write`. The ANSI standard version of `_formatted_write` is very large, and provides facilities not required in many applications. To reduce the memory consumption the following two alternative smaller versions are also provided in the standard C library:

`_medium_write`

As for `_formatted_write`, except that floating-point numbers are not supported. Any attempt to use a `%f`, `%g`, `%G`, `%e`, and `%E` specifier will produce the error:

```
FLOATS? wrong formatter installed!
```

`_medium_write` is considerably smaller than `_formatted_write`.

`_small_write`

As for `_medium_write`, except that it supports only the `%%`, `%d`, `%o`, `%c`, `%s` and `%x` specifiers for `int` objects, and does not support field width and precision arguments. The size of `_small_write` is 10–15% of the size of `_formatted_write`.

The default version is `_small_write`.

SELECTING THE WRITE FORMATTER VERSION

The selection of a write formatter is made in the XLINK control file. The default selection, `_small_write`, is made by the line:

```
-e_small_write=_formatted_write
```

To select the full ANSI version, remove this line.

To select `_medium_write`, replace this line with:

```
-e_medium_write=_formatted_write
```

REDUCED PRINTF

For many applications `sprintf` is not required, and even `printf` with `_small_write` provides more facilities than are justified by the memory consumed. Alternatively, a custom output routine may be required to support particular formatting needs and/or non-standard output devices.

For such applications, a highly reduced version of the entire `printf` function (without `sprintf`) is supplied in source form in the file `intwri.c`. This file can be modified to your requirements and the compiled module inserted into the library in place of the original using the procedure described for `putchar` above.

SCANF AND SSCANF

In a similar way to the `printf` and `sprintf` functions, `scanf` and `sscanf` use a common formatter called `_formatted_read`. The ANSI standard version of `_formatted_read` is very large, and provides facilities that are not required in many applications. To reduce the memory consumption, an alternative smaller version is also provided in the standard C library.

`_medium_read`

As for `_formatted_read`, except that no floating-point numbers are supported. `_medium_read` is considerably smaller than `_formatted_read`.

The default version is `_medium_read`.

SELECTING THE READ FORMATTER VERSION

The selection of a read formatter is made in the XLINK control file. The default selection, `_medium_read`, is made by the line:

```
-e_medium_read=_formatted_read
```

To select the full ANSI version, remove this line.

REGISTER I/O

A program may access the H8 I/O system using the memory-mapped internal special-function registers (SFRs).

All operators that apply to integral types except the unary `&` (address) operator may be applied to SFR registers. Predefined `define` declarations for the H8 family are supplied; see *Run-time library*, page 63.

HEAP SIZE

If the library functions `malloc` or `calloc` are used in the program, the C compiler creates a heap of memory from which their allocations are made. The default heap size is 2000 bytes.

The procedure for changing the heap size is described in the file `heap.c`.

INITIALIZATION

On processor reset, execution passes to a run-time system routine called `CSTARTUP`, which normally performs the following:

- ◆ Initializes the stack pointer.
- ◆ Initializes C file-level and static variables.
- ◆ Sets the CPU mode.
- ◆ Calls the user program function `main`.

`CSTARTUP` is also responsible for receiving and retaining control if the user program exits, whether through `exit` or `abort`.

VARIABLE AND I/O INITIALIZATION

In some applications you may want to initialize I/O registers, or omit the default initialization of data segments performed by `CSTARTUP`.

You can do this by providing a customized version of the routine `__low_level_init`, which is called from CSTARTUP before the data segments are initialized.

The value returned by `__low_level_init` determines whether data segments are initialized. The run-time library includes a dummy version of `__low_level_init` that simply returns 1, to cause CSTARTUP to initialize data segments.

The source of `__low_level_init` is provided in the file `lowinit.c`, by default located in the `icch8` directory. To perform your own I/O initializations, create a version of this routine containing the necessary code to do the initializations. If you also want to disable the initialization of data segments, make the routine return 0. Compile the customized routine and link it with the rest of your code.

MODIFYING CSTARTUP

If you want to modify CSTARTUP itself you will need to reassemble CSTARTUP with options which match your selected compilation options.

The overall procedure for assembling an appropriate copy of CSTARTUP is as follows:

- ◆ Make any required modifications to the assembler source of CSTARTUP, supplied by default in the file `c:\iar\icch8\cstartup.s37`, and save it under the same name.
- ◆ Assemble CSTARTUP using options that match your selected compilation options, as follows:

<i>Compilation option</i>	<i>Assembler option</i>
---------------------------	-------------------------

<code>-vn</code>	<code>-vn</code>
------------------	------------------

<code>-mx</code>	<code>-mx</code>
------------------	------------------

For example, if you have compiled for the H8/300H (`-v0`) and large memory model (`-m1`), you must assemble with the command:

```
ah8 cstartup -v0 -m1
```

This will create an object module file named `cstartup.r37`.

You should then use the following commands in the linker command file to make XLINK use the CSTARTUP module you have defined instead of the one in *library*:

```
-A cstartup  
-C library
```

DATA REPRESENTATION

This chapter describes how the H8 C Compiler represents each of the C data types, and gives recommendations for efficient coding.

DATA TYPES

The H8 C Compiler supports all ANSI C basic elements. Variables are stored with the least significant part located at the low memory address.

The following table gives the size and range of each C data type:

<i>Data type</i>	<i>Bytes</i>	<i>Range</i>	<i>Notes</i>
sfr	1	0 to 255	See the chapter <i>Extended keyword reference.</i>
sfrp	2	0 to 65535	
bit		0 to 1	
char (by default)	1	0 to 255	Equivalent to unsigned char
char (using -c option)	1	-128 to 127	Equivalent to signed char
signed char	1	-128 to 127	
unsigned char	1	0 to 255	
short, int	2	-2^{15} to $2^{15}-1$	-32768 to 32767
unsigned short, unsigned int	2	0 to $2^{16}-1$	0 to 65535
long	4	-2^{31} to $2^{31}-1$	-2147483648 to 2147483647
unsigned long	4	0 to $2^{32}-1$	0 to 4294967295
pointer	1 to 4		See the chapter <i>Extended keyword reference.</i>
float	4	$\pm 1.18\text{E}-38$ to $\pm 3.39\text{E}+38$	

<i>Data type</i>	<i>Bytes</i>	<i>Range</i>	<i>Notes</i>
double, long double	4	$\pm 1.18\text{E-}38$ to $\pm 3.39\text{E}+38$ (same as float)	
double, long double (using -2 option)	8	$\pm 2.23\text{E-}308$ to $\pm 1.79\text{E}+308$	

ENUM TYPE

The enum keyword creates each object with the shortest integer type (char, short, int, or long) required to contain its value.

CHAR TYPE

The char type is, by default, unsigned in the compiler, but the **Char is signed char** (-c) option allows you to make it signed. Note, however, that the library is compiled with char types as unsigned.

FLOATING POINT

Floating-point values are represented by either 4 or 8 byte numbers in standard IEEE format. Floating-point values below the smallest limit will be regarded as zero, and overflow gives undefined results.

The data types double and long double are normally equivalent to the float data type (4 bytes), but can be represented in an 8-byte double-precision format using the -2 option.

4-byte floating-point format

The memory layout of 4-byte floating-point numbers is:



The value of the number is:

$$(-1)^S * 2^{(\text{Exponent}-127)} * 1.\text{Mantissa}$$

Zero is represented by 4 bytes of zeros.

The precision of the float operators (+, -, *, and /) is approximately 7 decimal digits.

8-byte floating-point format

The memory layout of 8-byte floating-point numbers is:



The value of the number is:

$$(-1)^S * 2^{(\text{Exponent}-1023)} * 1.\text{Mantissa}$$

Zero is represented by 8 bytes of zeros.

The precision of the long double operators (+, -, *, and /) is approximately 16 decimal digits.

SPECIAL FUNCTION REGISTER VARIABLES

Special Function Register (*sfr*) variables allow a symbolic name to be associated with an address. They can be located anywhere in the 32-bit address space. They should be specified using full 32-bit sign-extended addresses, even in the small memory model. For example, an *sfr* at address 0xFFFF80 in -v0 must be specified as 0xFFFFFFFF80. This is done in the `ioxxx.h` include files.

If you want to access bits in the *sfr* or *sfrp* using the *bit_no* notation they must reside in one of the areas:

0x00000000 to 0x0FFFFFFF or 0xF0000000 to 0xFFFFFFFF.

This is only a restriction in the -v1 -m1 mode, and is due to the compiler implementation.

BITFIELDS

Bitfield unions and structures are extensions to ANSI C integer bitfields.

Bitfields in expressions will have the same data type as the base type (signed or unsigned char, short, int, or long).

By default bitfield variables are packed in elements of the specified type starting at the LSB position. Alternatively the bitfield packing can be reversed with the `#pragma bitfields=reversed` directive.

POINTERS

This section describes the H8 C Compiler's use of code pointers and data pointers.

CODE POINTERS

The code pointers are:

<i>Keyword</i>	<i>Storage in bytes</i>	<i>Restrictions</i>
<code>tiny_func</code>	1	May only point via the exception vector table.
<code>near_func</code>	2	Only in small memory model (-ms).
<code>far_func</code>	4	Only in large memory model (-ml).

The `tiny_func` pointer

A `tiny_func` pointer points to an exception vector. A `tiny_func` function call is to the address stored in the vector.

A function called by the `tiny_func` mechanism results in an indirect reference via the exception vector table. A `tiny_func` pointer may therefore only reference `tiny_func` functions.

The `near_func` pointer

The `near_func` pointer can only be used in the small memory model, and can access functions that is in the segment from 0x0 to 0xFFFF.

The `far_func` pointer

The `far_func` pointer can only be used in the large memory model, and gives unrestricted access to all functions.

Which of these pointer types is used as the default is determined by the memory model; see *Memory model*, page 64.

DATA POINTERS

The IAR C Compiler uses four different pointer types for data, ranging from the most efficient to the widest addressing range.

The data pointer types are:

<i>Keyword</i>	<i>Storage in bytes</i>	<i>Restrictions</i>
<code>tiny</code>	1	May only point into the <code>tiny</code> addressable area 0xFFFFFFFF00 to 0xFFFFFFFF.
<code>near</code>	2	-ms: none. -m1: may only point into the <code>near</code> addressable area 0xFFFF8000 to 0x00007FFF.
<code>far</code>	4	The referenced object must reside entirely in one 64 Kbyte segment.
<code>huge</code>	4	No restrictions.

Data in the short-addressable area can be accessed by `tiny` pointers. The benefit of `tiny` pointers is that they only require a single byte of storage.

The `far` pointer takes advantage of the fact that only the lower two bytes (the offset part) need to be considered in address arithmetic, but since the actual addressing uses four bytes, objects which are accessed through `far` pointers must reside entirely in one 64 Kbyte segment. In particular, this means that the object must not be larger than 64 Kbytes in size.

The `huge` pointer gives unrestricted access to all addresses.

Which of these pointer types is used as the default is determined by the memory model; see *Memory model*, page 64.

EFFICIENT CODING

It is important to appreciate the H8 architecture in order to avoid the use of inefficient language constructs. The following is a list of recommendations on how best to use the H8 C Compiler.

- ◆ Sensible use of the memory attributes (see the chapter *Extended keyword reference*) can enhance both speed and code size in critical applications.

- ◆ Bitfield types should be used only to conserve data memory space as they execute slowly on the H8. Use a bit mask on `unsigned char` or `unsigned int` instead of bitfields. If you must use bitfields, use unsigned for efficiency. Note, however, that `char` bitfields with one-bit bitfields are very efficient.
- ◆ Scalar variables that are not used outside their module should be declared as `static`, as this improves the possibility of temporarily keeping them in a register.
- ◆ Use as short data types as possible.
- ◆ Use unsigned data types, when possible. Sometimes unsigned operations execute more efficiently than the signed counterparts. This especially applies to division and modulo.
- ◆ Use ANSI prototypes. Function calls to ANSI functions are performed more efficiently than K&R-style functions; see the chapter *K&R and ANSI C language definitions*.
- ◆ Put static data in `tiny` and near address space as much as possible.

GENERAL C LIBRARY DEFINITIONS

This chapter gives an introduction to the C library functions, and summarizes them according to header file.

INTRODUCTION

The IAR C Compiler provides most of the important C library definitions that apply to PROM-based embedded systems. These are of three types:

- ◆ Standard C library definitions, for user programs. These are documented in this chapter.
- ◆ CSTARTUP, the single program module containing the start-up code.
- ◆ Intrinsic functions, allowing low-level use of H8S features.

LIBRARY OBJECT FILES

You must select the appropriate library object file for your chosen memory model and floating-point precision. See *Run-time library*, page 63, for more information. The linker includes only those routines that are required (directly or indirectly) by the user's program.

Most of the library definitions can be used without modification, that is, directly from the library object files supplied. There are some I/O-oriented routines (such as `putchar` and `getchar`) that you may need to customize for your target application.

The library object files are supplied having been compiled with the **Flag old-style functions** (`-gA`) option.

HEADER FILES

The user program gains access to library definitions through header files, which it incorporates using the `#include` directive. To avoid wasting time at compilation, the definitions are divided into a number of different header files each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do this can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY DEFINITIONS SUMMARY

This section lists the header files and summarizes the functions included in each. Header files may additionally contain target-specific definitions – these are documented in the chapter *Language extensions*.

CHARACTER HANDLING – `ctype.h`

<code>isalnum</code>	<code>int isalnum(int c)</code>	Letter or digit equality.
<code>isalpha</code>	<code>int isalpha(int c)</code>	Letter equality.
<code>iscntrl</code>	<code>int iscntrl(int c)</code>	Control code equality.
<code>isdigit</code>	<code>int isdigit(int c)</code>	Digit equality.
<code>isgraph</code>	<code>int isgraph(int c)</code>	Printable non-space character equality.
<code>islower</code>	<code>int islower(int c)</code>	Lower case equality.
<code>isprint</code>	<code>int isprint(int c)</code>	Printable character equality.
<code>ispunct</code>	<code>int ispunct(int c)</code>	Punctuation character equality.
<code>isspace</code>	<code>int isspace(int c)</code>	White-space character equality.
<code>isupper</code>	<code>int isupper(int c)</code>	Upper case equality.
<code>isxdigit</code>	<code>int isxdigit(int c)</code>	Hex digit equality.
<code>tolower</code>	<code>int tolower(int c)</code>	Converts to lower case.
<code>toupper</code>	<code>int toupper(int c)</code>	Converts to upper case.

LOW-LEVEL ROUTINES – icclbutl.h

<code>_formatted_read</code>	<code>int _formatted_read (const char **line, const char **format, va_list ap)</code>	Reads formatted data.
<code>_formatted_write</code>	<code>int _formatted_write (const char* format, void outputf (char, void *), void *sp, va_list ap)</code>	Formats and writes data.
<code>_medium_read</code>	<code>int _formatted_read (const char **line, const char **format, va_list ap)</code>	Reads formatted data excluding floating-point numbers.
<code>_medium_write</code>	<code>int _formatted_write (const char* format, void outputf (char, void *), void *sp, va_list ap)</code>	Writes formatted data excluding floating-point numbers.
<code>_small_write</code>	<code>int _formatted_write (const char* format, void outputf (char, void *), void *sp, va_list ap)</code>	Small formatted data write routine.

MATHEMATICS – math.h

<code>acos</code>	<code>double acos(double arg)</code>	Arc cosine.
<code>asin</code>	<code>double asin(double arg)</code>	Arc sine.
<code>atan</code>	<code>double atan(double arg)</code>	Arc tangent.
<code>atan2</code>	<code>double atan2(double arg1, double arg2)</code>	Arc tangent with quadrant.
<code>ceil</code>	<code>double ceil(double arg)</code>	Smallest integer greater than or equal to <i>arg</i> .
<code>cos</code>	<code>double cos(double arg)</code>	Cosine.
<code>cosh</code>	<code>double cosh(double arg)</code>	Hyperbolic cosine.
<code>exp</code>	<code>double exp(double arg)</code>	Exponential.
<code>exp10</code>	<code>double exp10(double arg)</code>	Ten to the power of.

GENERAL C LIBRARY DEFINITIONS

<code>fabs</code>	<code>double fabs(double arg)</code>	Double-precision floating-point absolute.
<code>floor</code>	<code>double floor(double arg)</code>	Largest integer less than or equal.
<code>fmod</code>	<code>double fmod(double arg1, double arg2)</code>	Floating-point remainder.
<code>frexp</code>	<code>double frexp(double arg1, int *arg2)</code>	Splits a floating-point number into two parts.
<code>ldexp</code>	<code>double ldexp(double arg1, int arg2)</code>	Multiply by power of two.
<code>log</code>	<code>double log(double arg)</code>	Natural logarithm.
<code>log10</code>	<code>double log10(double arg)</code>	Base-10 logarithm.
<code>modf</code>	<code>double modf(double value, double *iptr)</code>	Fractional and integer parts.
<code>pow</code>	<code>double pow(double arg1, double arg2)</code>	Raises to the power.
<code>sin</code>	<code>double sin(double arg)</code>	Sine.
<code>sinh</code>	<code>double sinh(double arg)</code>	Hyperbolic sine.
<code>sqrt</code>	<code>double sqrt(double arg)</code>	Square root.
<code>tan</code>	<code>double tan(double x)</code>	Tangent.
<code>tanh</code>	<code>double tanh(double arg)</code>	Hyperbolic tangent.

NON-LOCAL JUMPS – `setjmp.h`

<code>longjmp</code>	<code>void longjmp(jmp_buf env, int val)</code>	Long jump.
<code>setjmp</code>	<code>int setjmp(jmp_buf env)</code>	Sets up a jump return point.

VARIABLE ARGUMENTS – `stdarg.h`

<code>va_arg</code>	<code>type va_arg(va_list ap, mode)</code>	Next argument in function call.
<code>va_end</code>	<code>void va_end(va_list ap)</code>	Ends reading function call arguments.
<code>va_list</code>	<code>char *va_list[1]</code>	Argument list type.

<code>va_start</code>	<code>void va_start(va_list ap, parmN)</code>	Starts reading function call arguments.
INPUT/OUTPUT – <code>stdio.h</code>		
<code>getchar</code>	<code>int getchar(void)</code>	Gets character.
<code>gets</code>	<code>char *gets(char *s)</code>	Gets string.
<code>printf</code>	<code>int printf(const char *format, ...)</code>	Writes formatted data.
<code>putchar</code>	<code>int putchar(int value)</code>	Puts character.
<code>puts</code>	<code>int puts(const char *s)</code>	Puts string.
<code>scanf</code>	<code>int scanf(const char *format, ...)</code>	Reads formatted data.
<code>sprintf</code>	<code>int sprintf(char *s, const char *format,)</code>	Writes formatted data to a string.
<code>sscanf</code>	<code>int sscanf(const char *s, const char *format, ...)</code>	Reads formatted data from a string.
GENERAL UTILITIES – <code>stdlib.h</code>		
<code>abort</code>	<code>void abort(void)</code>	Terminates the program abnormally.
<code>abs</code>	<code>int abs(int j)</code>	Absolute value.
<code>atof</code>	<code>double atof(const char *nptr)</code>	Converts ASCII to double.
<code>atoi</code>	<code>int atoi(const char *nptr)</code>	Converts ASCII to int.
<code>atol</code>	<code>long atol(const char *nptr)</code>	Converts ASCII to long int.
<code>bsearch</code>	<code>void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compare) (const void *_key, const void *_base));</code>	Makes a generic search in an array.
<code>calloc</code>	<code>void *calloc(size_t nelem, size_t elsize)</code>	Allocates memory for an array of objects.

GENERAL C LIBRARY DEFINITIONS

div	div_t div(int <i>numer</i> , int <i>denom</i>)	Divide.
exit	void exit(int <i>status</i>)	Terminates the program.
free	void free(void * <i>ptr</i>)	Frees memory.
labs	long int labs(long int <i>j</i>)	Long absolute.
ldiv	ldiv_t ldiv(long int <i>numer</i> , long int <i>denom</i>)	Long division.
malloc	void *malloc(size_t <i>size</i>)	Allocates memory.
qsort	void qsort(const void * <i>base</i> , size_t <i>nmemb</i> , size_t <i>size</i> , int (* <i>compare</i>) (const void * <i>key</i> , const void * <i>_base</i>));	Makes a generic sort of an array.
rand	int rand(void)	Random number.
realloc	void *realloc(void * <i>ptr</i> , size_t <i>size</i>)	Reallocates memory.
srand	void srand(unsigned int <i>seed</i>)	Sets random number sequence.
strtod	double strtod(const char * <i>nptr</i> , char ** <i>endptr</i>)	Converts a string to double.
strtol	long int strtol(const char * <i>nptr</i> , char ** <i>endptr</i> , int <i>base</i>)	Converts a string to a long integer.
strtoul	unsigned long int strtoul (const char * <i>nptr</i> , char ** <i>endptr</i> , <i>base</i> int)	Converts a string to an unsigned long integer.

STRING HANDLING – string.h

memchr	void *memchr(const void * <i>s</i> , int <i>c</i> , size_t <i>n</i>)	Searches for a character in memory.
memcmp	int memcmp(const void * <i>s1</i> , const void * <i>s2</i> , size_t <i>n</i>)	Compares memory.
memcpy	void *memcpy(void * <i>s1</i> , const void * <i>s2</i> , size_t <i>n</i>)	Copies memory.

memmove	void *memmove(void *s1, const void *s2, size_t n)	Moves memory.
memset	void *memset(void *s, int c, size_t n)	Sets memory.
strcat	char *strcat(char *s1, const char *s2)	Concatenates strings.
strchr	char *strchr(const char *s, int c)	Searches for a character in a string.
strcmp	int strcmp(const char *s1, const char *s2)	Compares two strings.
strcoll	int strcoll(const char *s1, const char *s2)	Compares strings.
strcpy	char *strcpy(char *s1, const char *s2)	Copies string.
strcspn	size_t strcspn(const char *s1, const char *s2)	Spans excluded characters in string.
strerror	char *strerror(int <i>errnum</i>)	Gives an error message string.
strlen	size_t strlen(const char *s)	String length.
strncat	char *strncat(char *s1, const char *s2, size_t n)	Concatenates a specified number of characters with a string.
strncmp	int strncmp(const char *s1, const char *s2, size_t n)	Compares a specified number of characters with a string.
strncpy	char *strncpy(char *s1, const char *s2, size_t n)	Copies a specified number of characters from a string.
strpbrk	char *strpbrk(const char *s1, const char *s2)	Finds any one of specified characters in a string.
strrchr	char *strrchr(const char *s, int c)	Finds character from right of string.
strspn	size_t strspn(const char *s1, const char *s2)	Spans characters in a string.
strstr	char *strstr(const char *s1, const char *s2)	Searches for a substring.

strtok char *strtok(char *s1, const char *s2) Breaks a string into tokens.

strxfrm size_t strxfrm(char *s1, const char *s2, size_t n) Transforms a string and returns the length.

COMMON DEFINITIONS – stddef.h

No functions (various definitions including `size_t`, `NULL`, `ptrdiff_t`, `offsetof`, etc).

INTEGRAL TYPES – limits.h

No functions (various limits and sizes of integral types).

FLOATING-POINT TYPES – float.h

No functions (various limits and sizes of floating-point types).

ERRORS – errno.h

No functions (various error return values).

ASSERT – assert.h

assert void assert(int *expression*) Checks an expression.

C LIBRARY FUNCTIONS REFERENCE

This section gives an alphabetical list of the C library functions, with a full description of their operation, and the options available for each one.

The format of each function description is as follows:

	Function name	Header filename
Brief description	atoi	stdlib.h Converts ASCII to int.
Declaration		DECLARATION int atoi(const char *nptr)
Parameters		PARAMETERS <i>nptr</i> A pointer to a string containing a number in ASCII form.
Return value		RETURN VALUE The int number found in the string.
Description		DESCRIPTION Converts the ASCII string pointed to by <i>nptr</i> to an integer, skipping white space and terminating upon reaching any unrecognized character.
Examples		EXAMPLES " -3K" gives -3 "6" gives 6 "149" gives 149

FUNCTION NAME

The name of the C library function.

HEADER FILENAME

The function header filename.

BRIEF DESCRIPTION

A brief summary of the function.

DECLARATION

The C library declaration.

PARAMETERS

Details of each parameter in the declaration.

RETURN VALUE

The value, if any, returned by the function.

DESCRIPTION

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

EXAMPLES

One or more examples illustrating the function's use.

abort

`stdlib.h`

Terminates the program abnormally.

DECLARATION

```
void abort(void)
```

PARAMETERS

None.

RETURN VALUE

None.

DESCRIPTION

Terminates the program abnormally and does not return to the caller. This function calls the `exit` function, and by default the entry for this resides in `CSTARTUP`.

abs

stdlib.h

Absolute value.

DECLARATIONint abs(int *j*)**PARAMETERS***j* An int value.**RETURN VALUE**An int having the absolute value of *j*.**DESCRIPTION**Computes the absolute value of *j*.

acos

math.h

Arc cosine.

DECLARATIONdouble acos(double *arg*)**PARAMETERS***arg* A double in the range [-1,+1].**RETURN VALUE**The double arc cosine of *arg*, in the range [0, pi].**DESCRIPTION**Computes the principal value in radians of the arc cosine of *arg*.

asin

math.h

Arc sine.

DECLARATION

double asin(double *arg*)

PARAMETERS

arg A double in the range [-1,+1].

RETURN VALUE

The double arc sine of *arg*, in the range [-pi/2,+pi/2].

DESCRIPTION

Computes the principal value in radians of the arc sine of *arg*.

assert

assert.h

Checks an expression.

DECLARATION

void assert (int *expression*)

PARAMETERS

expression An expression to be checked.

RETURN VALUE

None.

DESCRIPTION

This is a macro that checks an expression. If it is false it prints a message to stderr and calls abort.

The message has the following format:

File *name*; line *num* # Assertion failure "*expression*"

To ignore assert calls put a `#define NDEBUG` statement before the `#include <assert.h>` statement.

atan

math.h

Arc tangent.

DECLARATION

double atan(double *arg*)

PARAMETERS

arg A double value.

RETURN VALUE

The double arc tangent of *arg*, in the range $[-\pi/2, \pi/2]$.

DESCRIPTION

Computes the arc tangent of *arg*.

atan2

math.h

Arc tangent with quadrant.

DECLARATION

double atan2(double *arg1*, double *arg2*)

PARAMETERS

arg1 A double value.

arg2 A double value.

RETURN VALUE

The double arc tangent of $arg1/arg2$, in the range $[-\pi, \pi]$.

DESCRIPTION

Computes the arc tangent of *arg1/arg2*, using the signs of both arguments to determine the quadrant of the return value.

atof

stdlib.h

Converts ASCII to double.

DECLARATION

double atof(const char **nptr*)

PARAMETERS

nptr A pointer to a string containing a number in ASCII form.

RETURN VALUE

The double number found in the string.

DESCRIPTION

Converts the string pointed to by *nptr* to a double-precision floating-point number, skipping white space and terminating upon reaching any unrecognized character.

EXAMPLES

" -3K" gives -3.00

".0006" gives 0.0006

"1e-4" gives 0.0001

atoi

stdlib.h

Converts ASCII to int.

DECLARATION

int atoi(const char **nptr*)

PARAMETERS

nptr A pointer to a string containing a number in ASCII form.

RETURN VALUE

The `int` number found in the string.

DESCRIPTION

Converts the ASCII string pointed to by *nptr* to an integer, skipping white space and terminating upon reaching any unrecognized character.

EXAMPLES

" -3K" gives -3

"6" gives 6

"149" gives 149

atol

`stdlib.h`

Converts ASCII to long int.

DECLARATION

```
long atol(const char *nptr)
```

PARAMETERS

nptr A pointer to a string containing a number in ASCII form.

RETURN VALUE

The long number found in the string.

DESCRIPTION

Converts the number found in the ASCII string pointed to by *nptr* to a long integer value, skipping white space and terminating upon reaching any unrecognized character.

EXAMPLES

" -3K" gives -3

"6" gives 6

"149" gives 149

bsearch

stdlib.h

Makes a generic search in an array.

DECLARATION

```
void *bsearch(const void *key, const void *base, size_t
nmemb, size_t size, int (*compare) (const void *_key,
const void *_base));
```

PARAMETERS

key Pointer to the searched for object.

base Pointer to the array to search.

nmemb Dimension of the array pointed to by *base*.

size Size of the array elements.

compare The comparison function which takes two arguments and returns:

< 0 (negative value) if *_key* is less than *_base*.

0 if *_key* equals *_base*.

> 0 (positive value) if *_key* is greater than *_base*.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer to the element of the array that matches the key.
Unsuccessful	Null.

DESCRIPTION

Searches an array of *nmemb* objects, pointed to by *base*, for an element that matches the object pointed to by *key*.

calloc

stdlib.h

Allocates memory for an array of objects.

DECLARATIONvoid *calloc(size_t *nelem*, size_t *elsize*)**PARAMETERS**

<i>nelem</i>	The number of objects.
<i>elsize</i>	A value of type size_t specifying the size of each object.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer to the start (lowest address) of the memory block.
Unsuccessful	Zero if there is no memory block of the required size or greater available.

DESCRIPTION

Allocates a memory block for an array of objects of the given size. To ensure portability, the size is not given in absolute units of memory such as bytes, but in terms of a size or sizes returned by the sizeof function.

The availability of memory depends on the default heap size.

ceil

math.h

Smallest integer greater than or equal to *arg*.**DECLARATION**double ceil(double *arg*)**PARAMETERS**

<i>arg</i>	A double value.
------------	-----------------

RETURN VALUE

A double having the smallest integral value greater than or equal to *arg*.

DESCRIPTION

Computes the smallest integral value greater than or equal to *arg*.

cos

math.h

Cosine.

DECLARATION

```
double cos(double arg)
```

PARAMETERS

arg A double value in radians.

RETURN VALUE

The double cosine of *arg*.

DESCRIPTION

Computes the cosine of *arg* radians.

cosh

math.h

Hyperbolic cosine.

DECLARATION

```
double cosh(double arg)
```

PARAMETERS

arg A double value in radians.

RETURN VALUE

The double hyperbolic cosine of *arg*.

DESCRIPTION

Computes the hyperbolic cosine of *arg* radians.

div

stdlib.h

Divide.

DECLARATION

```
div_t div(int numer, int denom)
```

PARAMETERS

numer The int numerator.

demon The int denominator.

RETURN VALUE

A structure of type `div_t` holding the quotient and remainder results of the division.

DESCRIPTION

Divides the numerator *numer* by the denominator *denom*. The type `div_t` is defined in `stdlib.h`.

If the division is inexact, the quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. The results are defined such that:

$$quot * denom + rem == numer$$

exit

stdlib.h

Terminates the program.

DECLARATION

void exit(int *status*)

PARAMETERS

status An int status value.

RETURN VALUE

None.

DESCRIPTION

Terminates the program normally. This function does not return to the caller. This function entry resides by default in CSTARTUP.

exp

math.h

Exponential.

DECLARATION

double exp(double *arg*)

PARAMETERS

arg A double value.

RETURN VALUE

A double with the value of the exponential function of *arg*.

DESCRIPTION

Computes the exponential function of *arg*; ie e^{arg} .

exp10

math.h

Exponential.

DECLARATIONdouble exp10(double *arg*)**PARAMETERS***arg* A double value.**RETURN VALUE**A double with the value of ten to the power of *arg*.**DESCRIPTION**Computes ten to the power of *arg*; ie 10^{arg} .

fabs

math.h

Double-precision floating-point absolute.

DECLARATIONdouble fabs(double *arg*)**PARAMETERS***arg* A double value.**RETURN VALUE**The double absolute value of *arg*.**DESCRIPTION**Computes the absolute value of the floating-point number *arg*.

floor

math.h

Largest integer less than or equal.

DECLARATION

double floor(double *arg*)

PARAMETERS

arg A double value.

RETURN VALUE

A double with the value of the largest integer less than or equal to *arg*.

DESCRIPTION

Computes the largest integral value less than or equal to *arg*.

fmod

math.h

Floating-point remainder.

DECLARATION

double fmod(double *arg1*, double *arg2*)

PARAMETERS

arg1 The double numerator.

arg2 The double denominator.

RETURN VALUE

The double remainder of the division *arg1/arg2*.

DESCRIPTION

Computes the remainder of *arg1/arg2*, ie the value *arg1 - i*arg2*, for some integer *i* such that, if *arg2* is non-zero, the result has the same sign as *arg1* and magnitude less than the magnitude of *arg2*.

free

stdlib.h

Frees memory.

DECLARATIONvoid free(void **ptr*)**PARAMETERS**

ptr A pointer to a memory block previously allocated by malloc, calloc, or realloc.

RETURN VALUE

None.

DESCRIPTION

Frees the memory used by the object pointed to by *ptr*. *ptr* must earlier have been assigned a value from malloc, calloc, or realloc.

frexp

math.h

Splits a floating-point number into two parts.

DECLARATIONdouble frexp(double *arg1*, int **arg2*)**PARAMETERS**

arg1 Floating-point number to be split.

arg2 Pointer to an integer to contain the exponent of *arg1*.

RETURN VALUE

The double mantissa of *arg1*, in the range 0.5 to 1.0.

DESCRIPTION

Splits the floating-point number *arg1* into an exponent stored in **arg2*, and a mantissa which is returned as the value of the function.

The values are as follows:

$$\text{mantissa} * 2^{\text{exponent}} = \text{value}$$

getchar

stdio.h

Gets character.

DECLARATION

int getchar(*void*)

PARAMETERS

None.

RETURN VALUE

An int with the ASCII value of the next character from the standard input stream.

DESCRIPTION

Gets the next character from the standard input stream.

You should customize this function for the particular target hardware configuration. The function is supplied in source format in the file `getchar.c`.

gets

stdio.h

Gets string.

DECLARATION

char *gets(char *s)

PARAMETERS

s A pointer to the string that is to receive the input.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer equal to <i>s</i> .
Unsuccessful	Null.

DESCRIPTION

Gets the next string from standard input and places it in the string pointed to. The string is terminated by end of line or end of file. The end-of-line character is replaced by zero.

This function calls `getchar`, which must be adapted for the particular target hardware configuration.

isalnum

`ctype.h`

Letter or digit equality.

DECLARATION

```
int isalnum(int c)
```

PARAMETERS

c An `int` representing a character.

RETURN VALUE

An `int` which is non-zero if *c* is a letter or digit, else zero.

DESCRIPTION

Tests whether a character is a letter or digit.

isalpha

ctype.h

Letter equality.

DECLARATION

```
int isalpha(int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if *c* is letter, else zero.

DESCRIPTION

Tests whether a character is a letter.

isctrl

ctype.h

Control code equality.

DECLARATION

```
int isctrl(int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if *c* is a control code, else zero.

DESCRIPTION

Tests whether a character is a control character.

isdigit

ctype.h

Digit equality.

DECLARATION

```
int isdigit(int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if *c* is a digit, else zero.

DESCRIPTION

Tests whether a character is a decimal digit.

isgraph

ctype.h

Printable non-space character equality.

DECLARATION

```
int isgraph(int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if *c* is a printable character other than space, else zero.

DESCRIPTION

Tests whether a character is a printable character other than space.

islower

cctype.h

Lower case equality.

DECLARATION

```
int islower(int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if *c* is lower case, else zero.

DESCRIPTION

Tests whether a character is a lower case letter.

isprint

cctype.h

Printable character equality.

DECLARATION

```
int isprint(int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if *c* is a printable character, including space, else zero.

DESCRIPTION

Tests whether a character is a printable character, including space.

ispunct

ctype.h

Punctuation character equality.

DECLARATION

```
int ispunct(int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if *c* is printable character other than space, digit, or letter, else zero.

DESCRIPTION

Tests whether a character is a printable character other than space, digit, or letter.

isspace

ctype.h

White-space character equality.

DECLARATION

```
int isspace (int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if *c* is a white-space character, else zero.

DESCRIPTION

Tests whether a character is a white-space character, that is, one of the following:

<i>Character</i>	<i>Symbol</i>
Space	' '
Formfeed	\f
Newline	\n
Carriage return	\r
Horizontal tab	\t
Vertical tab	\v

isupper

`cctype.h`

Upper case equality.

DECLARATION

```
int isupper(int c)
```

PARAMETERS

`c` An `int` representing a character.

RETURN VALUE

An `int` which is non-zero if `c` is upper case, else zero.

DESCRIPTION

Tests whether a character is an upper case letter.

isxdigit

ctype.h

Hex digit equality.

DECLARATION

```
int isxdigit(int c)
```

PARAMETERS

c An int representing a character.

RETURN VALUE

An int which is non-zero if *c* is a digit in upper or lower case, else zero.

DESCRIPTION

Tests whether the character is a hexadecimal digit in upper or lower case, that is, one of 0–9, a–f, or A–F.

labs

stdlib.h

Long absolute.

DECLARATION

```
long int labs(long int j)
```

PARAMETERS

j A long int value.

RETURN VALUE

The long int absolute value of *j*.

DESCRIPTION

Computes the absolute value of the long integer *j*.

ldexp

math.h

Multiply by power of two.

DECLARATION

```
double ldexp(double arg1, int arg2)
```

PARAMETERS

arg1 The double multiplier value.

arg2 The int power value.

RETURN VALUE

The double value of *arg1* multiplied by two raised to the power of *arg2*.

DESCRIPTION

Computes the value of the floating-point number multiplied by 2 raised to a power.

ldiv

stdlib.h

Long division

DECLARATION

```
ldiv_t ldiv(long int numer, long int denom)
```

PARAMETERS

numer The long int numerator.

denom The long int denominator.

RETURN VALUE

A struct of type `ldiv_t` holding the quotient and remainder of the division.

DESCRIPTION

Divides the numerator *numer* by the denominator *denom*. The type `ldiv_t` is defined in `stdlib.h`.

If the division is inexact, the quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. The results are defined such that:

$$quot * denom + rem == numer$$

log

`math.h`

Natural logarithm.

DECLARATION

`double log(double arg)`

PARAMETERS

arg A double value.

RETURN VALUE

The double natural logarithm of *arg*.

DESCRIPTION

Computes the natural logarithm of a number.

log10

`math.h`

Base-10 logarithm.

DECLARATION

`double log10(double arg)`

PARAMETERS

arg A double number.

longjmp

RETURN VALUE

The double base-10 logarithm of *arg*.

DESCRIPTION

Computes the base-10 logarithm of a number.

longjmp

setjmp.h

Long jump.

DECLARATION

```
void longjmp(jmp_buf env, int val)
```

PARAMETERS

env A struct of type jmp_buf holding the environment, set by setjmp.

val The int value to be returned by the corresponding setjmp.

RETURN VALUE

None.

DESCRIPTION

Restores the environment previously saved by setjmp. This causes program execution to continue as a return from the corresponding setjmp, returning the value *val*.

malloc

stdlib.h

Allocates memory.

DECLARATION

```
void *malloc(size_t size)
```

PARAMETERS

size A `size_t` object specifying the size of the object.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer to the start (lowest byte address) of the memory block.
Unsuccessful	Zero, if there is no memory block of the required size or greater available.

DESCRIPTION

Allocates a memory block for an object of the specified size.

The availability of memory depends on the size of the heap. For more information about changing the heap size refer to *Heap size*, page 69.

memchr

`string.h`

Searches for a character in memory.

DECLARATION

```
void *memchr(const void *s, int c, size_t n)
```

PARAMETERS

s A pointer to an object.

c An `int` representing a character.

n A value of type `size_t` specifying the size of each object.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer to the first occurrence of <i>c</i> in the <i>n</i> characters pointed to by <i>s</i> .
Unsuccessful	Null.

DESCRIPTION

Searches for the first occurrence of a character in a pointed-to region of memory of a given size.

Both the single character and the characters in the object are treated as unsigned.

memcmp

string.h

Compares memory.

DECLARATION

```
int memcmp(const void *s1, const void *s2, size_t n)
```

PARAMETERS

<i>s1</i>	A pointer to the first object.
<i>s2</i>	A pointer to the second object.
<i>n</i>	A value of type <code>size_t</code> specifying the size of each object.

RETURN VALUE

An integer indicating the result of comparison of the first *n* characters of the object pointed to by *s1* with the first *n* characters of the object pointed to by *s2*:

<i>Return value</i>	<i>Meaning</i>
>0	<i>s1</i> > <i>s2</i>
=0	<i>s1</i> = <i>s2</i>
<0	<i>s1</i> < <i>s2</i>

DESCRIPTION

Compares the first n characters of two objects.

memcpy

string.h

Copies memory.

DECLARATION

```
void *memcpy(void *s1, const void *s2, size_t n)
```

PARAMETERS

$s1$ A pointer to the destination object.

$s2$ A pointer to the source object.

n The number of characters to be copied.

RETURN VALUE

$s1$.

DESCRIPTION

Copies a specified number of characters from a source object to a destination object.

If the objects overlap, the result is undefined, so `memmove` should be used instead.

memmove

string.h

Moves memory.

DECLARATION

```
void *memmove(void *s1, const void *s2, size_t n)
```

PARAMETERS

- s1* A pointer to the destination object.
- s2* A pointer to the source object.
- n* The number of characters to be copied.

RETURN VALUE

s1.

DESCRIPTION

Copies a specified number of characters from a source object to a destination object.

Copying takes place as if the source characters are first copied into a temporary array that does not overlap either object, and then the characters from the temporary array are copied into the destination object.

memset

string.h

Sets memory.

DECLARATION

```
void *memset(void *s, int c, size_t n)
```

PARAMETERS

- s* A pointer to the destination object.
- c* An int representing a character.
- n* The size of the object.

RETURN VALUE

s.

DESCRIPTION

Copies a character (converted to an unsigned char) into each of the first specified number of characters of the destination object.

modf

math.h

Fractional and integer parts.

DECLARATION

```
double modf(double value, double *iptr)
```

PARAMETERS

value A double value.

iptr A pointer to the double that is to receive the integral part of *value*.

RETURN VALUEThe fractional part of *value*.**DESCRIPTION**

Computes the fractional and integer parts of *value*. The sign of both parts is the same as the sign of *value*.

pow

math.h

Raises to the power.

DECLARATION

```
double pow(double arg1, double arg2)
```

PARAMETERS

arg1 The double number.

arg2 The double power.

RETURN VALUE

arg1 raised to the power of *arg2*.

DESCRIPTION

Computes a number raised to a power.

printf

stdio.h

Writes formatted data.

DECLARATION

```
int printf(const char *format, ...)
```

PARAMETERS

format A pointer to the format string.

... The optional values that are to be printed under the control of *format*.

RETURN VALUE

<i>Result</i>	<i>Value</i>
---------------	--------------

Successful	The number of characters written.
------------	-----------------------------------

Unsuccessful	A negative value, if an error occurred.
--------------	---

DESCRIPTION

Writes formatted data to the standard output stream, returning the number of characters written or a negative value if an error occurred.

Since a complete formatter demands a lot of space there are several different formatters to choose. For more information see the chapter *Configuration*.

format is a string consisting of a sequence of characters to be printed and conversion specifications. Each conversion specification causes the next successive argument following the *format* string to be evaluated, converted, and written.

The form of a conversion specification is as follows:

```
% [flags] [field_width] [.precision] [length_modifier]  
conversion
```

Items inside [] are optional.

Flags

The *flags* are as follows:

<i>Flag</i>	<i>Effect</i>
-	Left adjusted field.
+	Signed values will always begin with plus or minus sign.
space	Values will always begin with minus or space.
#	Alternate form:
	<i>Specifier Effect</i>
octal	First digit will always be a zero.
G g	Decimal point printed and trailing zeros kept.
E e f	Decimal point printed.
X	Non-zero values prefixed with 0X.
x	Non-zero values prefixed with 0x.
0	Zero padding to field width (for d, i, o, u, x, X, e, E, f, g, and G specifiers).

Field width

The `field_width` is the number of characters to be printed in the field. The field will be padded with space if needed. A negative value indicates a left-adjusted field. A field width of `*` stands for the value of the next successive argument, which should be an integer.

Precision

The `precision` is the number of digits to print for integers (d, i, o, u, x, and X), the number of decimals printed for floating-point values (e, E, and f), and the number of significant digits for g and G conversions. A field width of `*` stands for the value of the next successive argument, which should be an integer.

Length modifier

The effect of each *length_modifier* is as follows:

<i>Length_modifier</i>	<i>Use</i>
h	before d, i, u, x, X, or o specifiers to denote a short int or unsigned short int value.
l	before d, i, u, x, X, or o specifiers to denote a long integer or unsigned long value.
L	before e, E, f, g, or G specifiers to denote a long double value.

Conversion

The result of each value of *conversion* is as follows:

<i>Conversion</i>	<i>Result</i>
d	Signed decimal value.
i	Signed decimal value.
o	Unsigned octal value.
u	Unsigned decimal value.
x	Unsigned hexadecimal value, using lower case (0-9, a-f).
X	Unsigned hexadecimal value, using upper case (0-9, A-F).
e	Double value in the style [-]d.ddde+dd.
E	Double value in the style [-]d.dddE+dd.
f	Double value in the style [-]ddd.ddd.
g	Double value in the style of f or e, whichever is the more appropriate.
G	Double value in the style of F or E, whichever is the more appropriate.
C	Single character constant.
s	String constant.

<i>Conversion</i>	<i>Result</i>
p	Pointer value (address).
n	No output, but store the number of characters written so far in the integer pointed to by the next argument.
%	% character.

Note that promotion rules convert all char and short int arguments to int while floats are converted to double.

printf calls the library function putchar, which must be adapted for the target hardware configuration.

The source of printf is provided in the file printf.c. The source of a reduced version that uses less program space and stack is provided in the file intwri.c.

EXAMPLES

After the following C statements:

```
int i=6, j=-6;
char *p = "ABC";
long l=100000;
float f1 = 0.0000001;
f2 = 750000;
double d = 2.2;
```

the effect of different printf function calls is shown in the following table; Δ represents space:

putchar

<i>Statement</i>	<i>Output</i>	<i>Characters output</i>
<code>printf("%c",p[1])</code>	B	1
<code>printf("%d",i)</code>	6	1
<code>printf("%3d",i)</code>	ΔΔ6	3
<code>printf("%.3d",i)</code>	006	3
<code>printf("%-10.3d",i)</code>	006ΔΔΔΔΔΔΔ	10
<code>printf("%10.3d",i)</code>	ΔΔΔΔΔΔΔ006	10
<code>printf("Value=%+3d",i)</code>	Value=Δ+6	9
<code>printf("%10.*d",i,j)</code>	ΔΔΔ-000006	10
<code>printf("String=[%s]",p)</code>	String=[ABC]	12
<code>printf("Value=%lX",l)</code>	Value=186A0	11
<code>printf("%f",f1)</code>	0.000000	8
<code>printf("%f",f2)</code>	750000.000000	13
<code>printf("%e",f1)</code>	1.000000e-07	12
<code>printf("%16e",d)</code>	ΔΔΔΔ2.200000e+00	16
<code>printf("%.4e",d)</code>	2.2000e+00	10
<code>printf("%g",f1)</code>	1e-07	5
<code>printf("%g",f2)</code>	750000	6
<code>printf("%g",d)</code>	2.2	3

putchar

stdio.h

Puts character.

DECLARATION

```
int putchar(int value)
```

PARAMETERS

value The int representing the character to be put.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	<i>value</i> .
Unsuccessful	The EOF macro.

DESCRIPTION

Writes a character to standard output.

You should customize this function for the particular target hardware configuration. The function is supplied in source format in the file `putchar.c`.

This function is called by `printf`.

puts

`stdio.h`

Puts string.

DECLARATION

```
int puts(const char *s)
```

PARAMETERS

s A pointer to the string to be put.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A non-negative value.
Unsuccessful	-1 if an error occurred.

DESCRIPTION

Writes a string followed by a new-line character to the standard output stream.

qsort

stdlib.h

Makes a generic sort of an array.

DECLARATION

```
void qsort (const void *base, size_t nmemb, size_t size,  
int (*compare) (const void *_key, const void *_base));
```

PARAMETERS

- base* Pointer to the array to sort.
- nmemb* Dimension of the array pointed to by *base*.
- size* Size of the array elements.
- compare* The comparison function, which takes two arguments and returns:
 - < 0 (negative value) if *_key* is less than *_base*.
 - 0 if *_key* equals *_base*.
 - > 0 (positive value) if *_key* is greater than *_base*.

RETURN VALUE

None.

DESCRIPTION

Sorts an array of *nmemb* objects pointed to by *base*.

rand

stdlib.h

Random number.

DECLARATION

```
int rand(void)
```

PARAMETERS

None.

RETURN VALUE

The next `int` in the random number sequence.

DESCRIPTION

Computes the next in the current sequence of pseudo-random integers, converted to lie in the range `[0, RAND_MAX]`.

See `srand` for a description of how to seed the pseudo-random sequence.

realloc

`stdlib.h`

Reallocates memory.

DECLARATION

```
void *realloc(void *ptr, size_t size)
```

PARAMETERS

ptr A pointer to the start of the memory block.

size A value of type `size_t` specifying the size of the object.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer to the start (lowest address) of the memory block.
Unsuccessful	Null, if no memory block of the required size or greater was available.

DESCRIPTION

Changes the size of a memory block (which must be allocated by `malloc`, `calloc`, or `realloc`).

scanf

stdio.h

Reads formatted data.

DECLARATION

```
int scanf(const char *format, ...)
```

PARAMETERS

format A pointer to a format string.

... Optional pointers to the variables that are to receive values.

RETURN VALUE

<i>Result</i>	<i>Value</i>
---------------	--------------

Successful	The number of successful conversions.
------------	---------------------------------------

Unsuccessful	-1 if the input was exhausted.
--------------	--------------------------------

DESCRIPTION

Reads formatted data from standard input.

Since a complete formatter demands a lot of space there are several different formatters to choose. For more information see *Input and output*, page 66.

format is a string consisting of a sequence of ordinary characters and conversion specifications. Each ordinary character reads a matching character from the input. Each conversion specification accepts input meeting the specification, converts it, and assigns it to the object pointed to by the next successive argument following *format*.

If the format string contains white-space characters, input is scanned until a non-white-space character is found.

The form of a conversion specification is as follows:

```
% [assign_suppress] [field_width] [length_modifier]  
conversion
```

Items inside [] are optional.

Assign suppress

If a `*` is included in this position, the field is scanned but no assignment is carried out.

field_width

The `field_width` is the maximum field to be scanned. The default is until no match occurs.

length_modifier

The effect of each `length_modifier` is as follows:

<i>Length modifier</i>	<i>Before</i>	<i>Meaning</i>
l	d, i, or n	long int as opposed to int.
	o, u, or x	unsigned long int as opposed to unsigned int.
	e, E, g, G, or f	double operand as opposed to float.
h	d, i, or n	short int as opposed to int.
	o, u, or x	unsigned short int as opposed to unsigned int.
L	e, E, g, G, or f	long double operand as opposed to float.

Conversion

The meaning of each conversion is as follows:

<i>Conversion</i>	<i>Meaning</i>
d	Optionally signed decimal integer value.
i	Optionally signed integer value in standard C notation, that is, is decimal, octal (0n) or hexadecimal (0xn, 0Xn).
o	Optionally signed octal integer.
u	Unsigned decimal integer.
x	Optionally signed hexadecimal integer.
X	Optionally signed hexadecimal integer (equivalent to x).
f	Floating-point constant.

<i>Conversion</i>	<i>Meaning</i>
e E g G	Floating-point constant (equivalent to f).
s	Character string.
c	One or <code>field_width</code> characters.
n	No read, but store number of characters read so far in the integer pointed to by the next argument.
p	Pointer value (address).
[Any number of characters matching any of the characters before the terminating <code>]</code> . For example, <code>[abc]</code> means a, b, or c.
[]	Any number of characters matching <code>]</code> or any of the characters before the further, terminating <code>]</code> . For example, <code>[]abc]</code> means <code>]</code> , a, b, or c.
[^	Any number of characters not matching any of the characters before the terminating <code>]</code> . For example, <code>[^abc]</code> means not a, b, or c.
[^]	Any number of characters not matching <code>]</code> or any of the characters before the further, terminating <code>]</code> . For example, <code>[^]abc]</code> means not <code>]</code> , a, b, or c.
%	% character.

In all conversions except `c`, `n`, and all varieties of `[`, leading white-space characters are skipped.

`scanf` indirectly calls `get char`, which must be adapted for the actual target hardware configuration.

EXAMPLES

For example, after the following program:

```
int n, i;  
char name[50];  
float x;  
n = scanf("%d%f%s", &i, &x, name)
```

this input line:

```
25 54.32E-1 Hello World
```

will set the variables as follows:

```
n = 3, i = 25, x = 5.432, name="Hello World"
```

and this function:

```
scanf("%2d%f%d %[0123456789]", &i, &x, name)
```

with this input line:

```
56789 0123 56a72
```

will set the variables as follows:

```
i = 56, x = 789.0, name="56" (0123 unassigned)
```

setjmp

setjmp.h

Sets up a jump return point.

DECLARATION

```
int setjmp(jmp_buf env)
```

PARAMETERS

env An object of type `jmp_buf` into which `setjmp` is to store the environment.

RETURN VALUE

Zero.

Execution of a corresponding `longjmp` causes execution to continue as if it was a return from `setjmp`, in which case the value of the `int` value given in the `longjmp` is returned.

DESCRIPTION

Saves the environment in *env* for later use by `longjmp`.

Note that `setjmp` must always be used in the same function or at a higher nesting level than the corresponding call to `longjmp`.

sin

sin

math.h

Sine.

DECLARATION

double sin(double *arg*)

PARAMETERS

arg A double value in radians.

RETURN VALUE

The double sine of *arg*.

DESCRIPTION

Computes the sine of a number.

sinh

math.h

Hyperbolic sine.

DECLARATION

double sinh(double *arg*)

PARAMETERS

arg A double value in radians.

RETURN VALUE

The double hyperbolic sine of *arg*.

DESCRIPTION

Computes the hyperbolic sine of *arg* radians.

printf

stdio.h

Writes formatted data to a string.

DECLARATION

```
int printf(char *s, const char *format, ...)
```

PARAMETERS

<i>s</i>	A pointer to the string that is to receive the formatted data.
<i>format</i>	A pointer to the format string.
...	The optional values that are to be printed under the control of <i>format</i> .

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	The number of characters written.
Unsuccessful	A negative value if an error occurred.

DESCRIPTION

Operates exactly as `printf` except the output is directed to a string. See `printf` for details.

`printf` does not use the function `putchar`, and therefore can be used even if `putchar` is not available for the target configuration.

Since a complete formatter demands a lot of space there are several different formatters to choose. For more information see *Input and output*, page 66.

sqrt

math.h

Square root.

DECLARATION

```
double sqrt(double arg)
```

PARAMETERS

arg A double value.

RETURN VALUE

The double square root of *arg*.

DESCRIPTION

Computes the square root of a number.

srand

stdlib.h

Sets random number sequence.

DECLARATION

void srand(unsigned int *seed*)

PARAMETERS

seed An unsigned int value identifying the particular random number sequence.

RETURN VALUE

None.

DESCRIPTION

Selects a repeatable sequence of pseudo-random numbers.

The function rand is used to get successive random numbers from the sequence. If rand is called before any calls to srand have been made, the sequence generated is that which is generated after srand(1).

sscanf

stdio.h

Reads formatted data from a string.

DECLARATION

int sscanf(const char *s, const char *format, ...)

PARAMETERS

s A pointer to the string containing the data.

format A pointer to a format string.

... Optional pointers to the variables that are to receive values.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	The number of successful conversions.
Unsuccessful	-1 if the input was exhausted.

DESCRIPTION

Operates exactly as scanf except the input is taken from the string s. See scanf, for details.

The function sscanf does not use getchar, and so can be used even when getchar is not available for the target configuration.

Since a complete formatter demands a lot of space there are several different formatters to choose. For more information see the chapter *Configuration*.

strcat

string.h

Concatenates strings.

DECLARATION

char *strcat(char *s1, const char *s2)

PARAMETERS

- s1* A pointer to the first string.
- s2* A pointer to the second string.

RETURN VALUE

s1.

DESCRIPTION

Appends a copy of the second string to the end of the first string. The initial character of the second string overwrites the terminating null character of the first string.

strchr

string.h

Searches for a character in a string.

DECLARATION

```
char *strchr(const char *s, int c)
```

PARAMETERS

- c* An int representation of a character.
- s* A pointer to a string.

RETURN VALUE

If successful, a pointer to the first occurrence of *c* (converted to a char) in the string pointed to by *s*.

If unsuccessful due to *c* not being found, null.

DESCRIPTION

Finds the first occurrence of a character (converted to a char) in a string. The terminating null character is considered to be part of the string.

strcmp

string.h

Compares two strings.

DECLARATION

```
int strcmp(const char *s1, const char *s2)
```

PARAMETERS

s1 A pointer to the first string.

s2 A pointer to the second string.

RETURN VALUE

The int result of comparing the two strings:

<i>Return value</i>	<i>Meaning</i>
---------------------	----------------

>0	<i>s1</i> > <i>s2</i>
----	-----------------------

=0	<i>s1</i> = <i>s2</i>
----	-----------------------

<0	<i>s1</i> < <i>s2</i>
----	-----------------------

DESCRIPTION

Compares the two strings.

strcoll

string.h

Compares strings.

DECLARATION

```
int strcoll(const char *s1, const char *s2)
```

PARAMETERS

s1 A pointer to the first string.

s2 A pointer to the second string.

RETURN VALUE

The `int` result of comparing the two strings:

<i>Return value</i>	<i>Meaning</i>
<code>>0</code>	<code>s1 > s2</code>
<code>=0</code>	<code>s1 = s2</code>
<code><0</code>	<code>s1 < s2</code>

DESCRIPTION

Compares the two strings. This function operates identically to `strcmp` and is provided for compatibility only.

strcpy

`string.h`

Copies string.

DECLARATION

```
char *strcpy(char *s1, const char *s2)
```

PARAMETERS

`s1` A pointer to the destination object.

`s2` A pointer to the source string.

RETURN VALUE

`s1`.

DESCRIPTION

Copies a string into an object.

strcspn

string.h

Spans excluded characters in string.

DECLARATIONsize_t strcspn(const char **s1*, const char **s2*)**PARAMETERS***s1* A pointer to the subject string.*s2* A pointer to the object string.**RETURN VALUE**

The int length of the maximum initial segment of the string pointed to by *s1* that consists entirely of characters *not* from the string pointed to by *s2*.

DESCRIPTION

Finds the maximum initial segment of a subject string that consists entirely of characters *not* from an object string.

strerror

string.h

Gives an error message string.

DECLARATIONchar * strerror (int *errnum*)**PARAMETERS***errnum* The error message to return.**RETURN VALUE**

strerror is an implementation-defined function. In the H8 C Compiler it returns the following strings.

strlen

<i>errno</i>	<i>String returned</i>
EZERO	"no error"
EDOM	"domain error"
ERANGE	"range error"
<code>errno < 0 errno > Max_err_num</code>	"unknown error"
All other numbers	"error No. <i>errno</i> "

DESCRIPTION

Returns an error message string.

strlen

string.h

String length.

DECLARATION

```
size_t strlen(const char *s)
```

PARAMETERS

s A pointer to a string.

RETURN VALUE

An object of type `size_t` indicating the length of the string.

DESCRIPTION

Finds the number of characters in a string, not including the terminating null character.

strncat

string.h

Concatenates a specified number of characters with a string.

DECLARATION

```
char *strncat(char *s1, const char *s2, size_t n)
```

PARAMETERS

s1 A pointer to the destination string.
s2 A pointer to the source string.
n The number of characters of the source string to use.

RETURN VALUE

s1.

DESCRIPTION

Appends not more than *n* initial characters from the source string to the end of the destination string.

strncmp

string.h

Compares a specified number of characters with a string.

DECLARATION

```
int strncmp(const char *s1, const char *s2, size_t n)
```

PARAMETERS

s1 A pointer to the first string.
s2 A pointer to the second string.
n The number of characters of the source string to compare.

RETURN VALUE

The int result of the comparison of not more than *n* initial characters of the two strings:

<i>Return value</i>	<i>Meaning</i>
>0	<i>s1</i> > <i>s2</i>
=0	<i>s1</i> = <i>s2</i>
<0	<i>s1</i> < <i>s2</i>

DESCRIPTION

Compares not more than *n* initial characters of the two strings.

strncpy

string.h

Copies a specified number of characters from a string.

DECLARATION

char *strncpy(char *s1, const char *s2, size_t n)

PARAMETERS

s1 A pointer to the destination object.

s2 A pointer to the source string.

n The number of characters of the source string to copy.

RETURN VALUE

s1.

DESCRIPTION

Copies not more than *n* initial characters from the source string into the destination object.

strpbrk

string.h

Finds any one of specified characters in a string.

DECLARATION

char *strpbrk(const char *s1, const char *s2)

PARAMETERS

s1 A pointer to the subject string.

s2 A pointer to the object string.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer to the first occurrence in the subject string of any character from the object string.
Unsuccessful	Null if none were found.

DESCRIPTION

Searches one string for any occurrence of any character from a second string.

strrchr

string.h

Finds character from right of string.

DECLARATION

```
char *strrchr(const char *s, int c)
```

PARAMETERS

s A pointer to a string.

c An int representing a character.

RETURN VALUE

If successful, a pointer to the last occurrence of *c* in the string pointed to by *s*.

DESCRIPTION

Searches for the last occurrence of a character (converted to a char) in a string. The terminating null character is considered to be part of the string.

strspn

string.h

Spans characters in a string.

DECLARATION

```
size_t strspn(const char *s1, const char *s2)
```

PARAMETERS

s1 A pointer to the subject string.

s2 A pointer to the object string.

RETURN VALUE

The length of the maximum initial segment of the string pointed to by *s1* that consists entirely of characters from the string pointed to by *s2*.

DESCRIPTION

Finds the maximum initial segment of a subject string that consists entirely of characters from an object string.

strstr

string.h

Searches for a substring.

DECLARATION

```
char *strstr(const char *s1, const char *s2)
```

PARAMETERS

s1 A pointer to the subject string.

s2 A pointer to the object string.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	A pointer to the first occurrence in the string pointed to by <i>s1</i> of the sequence of characters (excluding the terminating null character) in the string pointed to by <i>s2</i> .
Unsuccessful	Null if the string was not found. <i>s1</i> if <i>s2</i> is pointing to a string with zero length.

DESCRIPTION

Searches one string for an occurrence of a second string.

strtod

stdlib.h

Converts a string to double.

DECLARATION

```
double strtod(const char *nptr, char **endptr)
```

PARAMETERS

<i>nptr</i>	A pointer to a string.
<i>endptr</i>	A pointer to a pointer to a string.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	The double result of converting the ASCII representation of an floating-point constant in the string pointed to by <i>nptr</i> , leaving <i>endptr</i> pointing to the first character after the constant.
Unsuccessful	Zero, leaving <i>endptr</i> indicating the first non-space character.

DESCRIPTION

Converts the ASCII representation of a number into a `double`, stripping any leading white space.

strtok

`string.h`

Breaks a string into tokens.

DECLARATION

```
char *strtok(char *s1, const char *s2)
```

PARAMETERS

s1 A pointer to a string to be broken into tokens.

s2 A pointer to a string of delimiters.

RETURN VALUE

<i>Result</i>	<i>Value</i>
---------------	--------------

Successful	A pointer to the token.
------------	-------------------------

Unsuccessful	Zero.
--------------	-------

DESCRIPTION

Finds the next token in the string *s1*, separated by one or more characters from the string of delimiters *s2*.

The first time you call `strtok`, *s1* should be the string you want to break into tokens. `strtok` saves this string. On each subsequent call, *s1* should be `NULL`. `strtok` searches for the next token in the string it saved. *s2* can be different from call to call.

If `strtok` finds a token, it returns a pointer to the first character in it. Otherwise it returns `NULL`. If the token is not at the end of the string, `strtok` replaces the delimiter with a null character (`\0`).

strtol

stdlib.h

Converts a string to a long integer.

DECLARATION

```
long int strtol(const char *nptr, char **endptr, int
base)
```

PARAMETERS

nptr A pointer to a string.
endptr A pointer to a pointer to a string.
base An *int* value specifying the base.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	The long int result of converting the ASCII representation of an integer constant in the string pointed to by <i>nptr</i> , leaving <i>endptr</i> pointing to the first character after the constant.
Unsuccessful	Zero, leaving <i>endptr</i> indicating the first non-space character.

DESCRIPTION

Converts the ASCII representation of a number into a long int using the specified base, and stripping any leading white space.

If the base is zero the sequence expected is an ordinary integer. Otherwise the expected sequence consists of digits and letters representing an integer with the radix specified by *base* (must be between 2 and 36). The letters [a, z] and [A, Z] are ascribed the values 10 to 35. If the base is 16, the 0x portion of a hex integer is allowed as the initial sequence.

strtoul

stdlib.h

Converts a string to an unsigned long integer.

DECLARATION

```
unsigned long int strtoul(const char *nptr,  
char **endptr, base int)
```

PARAMETERS

nptr A pointer to a string
endptr A pointer to a pointer to a string
base An int value specifying the base.

RETURN VALUE

<i>Result</i>	<i>Value</i>
Successful	The unsigned long int result of converting the ASCII representation of an integer constant in the string pointed to by <i>nptr</i> , leaving <i>endptr</i> pointing to the first character after the constant.
Unsuccessful	Zero, leaving <i>endptr</i> indicating the first non-space character.

DESCRIPTION

Converts the ASCII representation of a number into an unsigned long int using the specified base, stripping any leading white space.

If the base is zero the sequence expected is an ordinary integer. Otherwise the expected sequence consists of digits and letters representing an integer with the radix specified by *base* (must be between 2 and 36). The letters [a, z] and [A, Z] are ascribed the values 10 to 35. If the base is 16, the 0x portion of a hex integer is allowed as the initial sequence.

strxfrm

string.h

Transforms a string and returns the length.

DECLARATION

```
size_t strxfrm(char *s1, const char *s2, size_t n)
```

PARAMETERS

s1 Return location of the transformed string.

s2 String to transform.

n Maximum number of characters to be placed in *s1*.

RETURN VALUE

The length of the transformed string, not including the terminating null character.

DESCRIPTION

The transformation is such that if the `strcmp` function is applied to two transformed strings, it returns a value corresponding to the result of the `strcoll` function applied to the same two original strings.

tan

math.h

Tangent.

DECLARATION

```
double tan(double arg)
```

PARAMETERS

arg A double value in radians.

RETURN VALUE

The double tangent of *arg*.

DESCRIPTION

Computes the tangent of *arg* radians.

tanh

math.h

Hyperbolic tangent.

DECLARATION

```
double tanh(double arg)
```

PARAMETERS

arg A double value in radians.

RETURN VALUE

The double hyperbolic tangent of *arg*.

DESCRIPTION

Computes the hyperbolic tangent of *arg* radians.

tolower

ctype.h

Converts to lower case.

DECLARATION

```
int tolower(int c)
```

PARAMETERS

c The int representation of a character.

RETURN VALUE

The int representation of the lower case character corresponding to *c*.

DESCRIPTION

Converts a character into lower case.

toupper

ctype.h

Converts to upper case.

DECLARATION

int toupper(int c)

PARAMETERS*c* The int representation of a character.**RETURN VALUE**The int representation of the upper case character corresponding to *c*.**DESCRIPTION**Converts a character into upper case.

va_arg

stdarg.h

Next argument in function call.

DECLARATIONtype va_arg(va_list *ap*, *mode*)**PARAMETERS***ap* A value of type `va_list`.*mode* A type name such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a `*` to type.**RETURN VALUE**See below.

DESCRIPTION

A macro that expands to an expression with the type and value of the next argument in the function call. After initialization by `va_start`, this is the argument after that specified by `parmN`. `va_arg` advances `ap` to deliver successive arguments in order.

For an example of the use of `va_arg` and associated macros, see the files `printf.c` and `intwri.c`.

va_end

`stdarg.h`

Ends reading function call arguments.

DECLARATION

```
void va_end(va_list ap)
```

PARAMETERS

ap A pointer of type `va_list` to the variable-argument list.

RETURN VALUE

See below.

DESCRIPTION

A macro that facilitates normal return from the function whose variable argument list was referenced by the expansion `va_start` that initialized `va_list ap`.

va_list

`stdarg.h`

Argument list type.

DECLARATION

```
char *va_list[1]
```


PARAMETERS

None.

RETURN VALUE

See below.

DESCRIPTION

An array type suitable for holding information needed by `va_arg` and `va_end`.

va_start

`stdarg.h`

Starts reading function call arguments.

DECLARATION

```
void va_start(va_list ap, parmN)
```

PARAMETERS

ap A pointer of type `va_list` to the variable-argument list.

parmN The identifier of the rightmost parameter in the variable parameter list in the function definition.

RETURN VALUE

See below.

DESCRIPTION

A macro that initializes *ap* for use by `va_arg` and `va_end`.

`_formatted_read`

`icclbut1.h`

Reads formatted data.

DECLARATION

```
int _formatted_read (const char **line, const char
**format, va_list ap)
```

PARAMETERS

line A pointer to a pointer to the data to scan.

format A pointer to a pointer to a standard `scanf` format specification string.

ap A pointer of type `va_list` to the variable argument list.

RETURN VALUE

The number of successful conversions.

DESCRIPTION

Reads formatted data. This function is the basic formatter of `scanf`.

`_formatted_read` is concurrently reusable (reentrant).

Note that the use of `_formatted_read` requires the special ANSI-defined macros in the file `stdarg.h`, described above. In particular:

- ◆ There must be a variable `ap` of type `va_list`.
- ◆ There must be a call to `va_start` before calling `_formatted_read`.
- ◆ There must be a call to `va_end` before leaving the current context.
- ◆ The argument to `va_start` must be the formal parameter immediately to the left of the variable argument list.

`_formatted_write`

icclbut1.h

Formats and writes data.

DECLARATION

```
int _formatted_write (const char *format, void outputf  
(char, void *), void *sp, va_list ap)
```

PARAMETERS

- | | |
|----------------|---|
| <i>format</i> | A pointer to standard printf/sprintf format specification string. |
| <i>outputf</i> | A function pointer to a routine that actually writes a single character created by <code>_formatted_write</code> . The first parameter to this function contains the actual character value and the second a pointer whose value is always equivalent to the third parameter of <code>_formatted_write</code> . |
| <i>sp</i> | A pointer to some type of data structure that the low-level output function may need. If there is no need for anything more than just the character value, this parameter must still be specified with <code>(void *) 0</code> as well as declared in the output function. |
| <i>ap</i> | A pointer of type <code>va_list</code> to the variable-argument list. |

RETURN VALUE

The number of characters written.

DESCRIPTION

Formats write data. This function is the basic formatter of `printf` and `sprintf`, but through its universal interface can easily be adapted for writing to non-standard display devices.

Since a complete formatter demands a lot of space there are several different formatters to choose. For more information see the chapter *Configuration*.

`_formatted_write` is concurrently reusable (reentrant).

Note that the use of `_formatted_write` requires the special ANSI-defined macros in the file `stdarg.h`, described above. In particular:

- ◆ There must be a variable *ap* of type `va_list`.
- ◆ There must be a call to `va_start` before calling `_formatted_write`.
- ◆ There must be a call to `va_end` before leaving the current context.
- ◆ The argument to `va_start` must be the formal parameter immediately to the left of the variable argument list.

For an example of how to use `_formatted_write`, see the file `printf.c`.

`_medium_read`

`icclbut1.h`

Reads formatted data excluding floating-point numbers.

DECLARATION

```
int _medium_read (const char **line, const char **format,  
va_list ap)
```

PARAMETERS

- | | |
|---------------|--|
| <i>line</i> | A pointer to a pointer to the data to scan. |
| <i>format</i> | A pointer to a pointer to a standard <code>scanf</code> format specification string. |
| <i>ap</i> | A pointer of type <code>va_list</code> to the variable argument list. |

RETURN VALUE

The number of successful conversions.

DESCRIPTION

A reduced version of `_formatted_read` which is half the size, but does not support floating-point numbers.

For further information see `_formatted_read`.

`_medium_write`

icclbut1.h

Writes formatted data excluding floating-point numbers.

DECLARATION

```
int _medium_write (const char *format, void outputf(char,  
void *), void *sp, va_list ap)
```

PARAMETERS

- | | |
|----------------|---|
| <i>format</i> | A pointer to standard printf/sprintf format specification string. |
| <i>outputf</i> | A function pointer to a routine that actually writes a single character created by <code>_formatted_write</code> . The first parameter to this function contains the actual character value and the second a pointer whose value is always equivalent to the third parameter of <code>_formatted_write</code> . |
| <i>sp</i> | A pointer to some type of data structure that the low-level output function may need. If there is no need for anything more than just the character value, this parameter must still be specified with <code>(void *) 0</code> as well as declared in the output function. |
| <i>ap</i> | A pointer of type <code>va_list</code> to the variable-argument list. |

RETURN VALUE

The number of characters written.

DESCRIPTION

A reduced version of `_formatted_write` which is half the size, but does not support floating-point numbers.

For further information see `_formatted_write`.

`_small_write`

`icclbut1.h`

Small formatted data write routine.

DECLARATION

```
int _small_write (const char *format, void outputf (char,  
void *), void *sp, va_list ap)
```

PARAMETERS

- format* A pointer to standard printf/sprintf format specification string.
- outputf* A function pointer to a routine that actually writes a single character created by `_formatted_write`. The first parameter to this function contains the actual character value and the second a pointer whose value is always equivalent to the third parameter of `_formatted_write`.
- sp* A pointer to some type of data structure that the low-level output function may need. If there is no need for anything more than just the character value, this parameter must still be specified with `(void *) 0` as well as declared in the output function.
- ap* A pointer of type `va_list` to the variable-argument list.

RETURN VALUE

The number of characters written.

DESCRIPTION

This is a small version of `_formatted_write` which is about a quarter of the size.

The `_small_write` formatter supports only the following specifiers for `int` objects:

`%%`, `%d`, `%o`, `%c`, `%s`, and `%x`

It does not support field width or precision arguments, and no diagnostics will be produced if unsupported specifiers or modifiers are used.

For further information see `_formatted_write`.

LANGUAGE EXTENSIONS

This chapter summarizes the extensions provided in the H8 C Compiler to support specific features of the H8 microprocessor.

INTRODUCTION

The extensions are provided in three ways:

- ◆ As extended keywords. By default, the compiler conforms to the ANSI specifications and H8 extensions are not available. The command line option `-e` makes the extended keywords available, and hence reserves them so that they cannot be used as variable names.
- ◆ As `#pragma` keywords. These provide `#pragma` directives which control how the compiler allocates memory, whether the compiler allows extended keywords, and whether the compiler outputs warning messages.
- ◆ As intrinsic functions. These provide direct access to very low-level processor details.

EXTENDED KEYWORDS SUMMARY

The extended keywords provide the following facilities:

BIT VARIABLES

The program may take advantage of the H8 bit-addressing modes by using the following data type:

`bit`

ADDRESSING CONTROL

By default the address range in which the compiler places a variable is determined by the memory model chosen. The program may achieve additional efficiency for special cases by overriding the default by using one of the storage modifiers:

`tiny`, `near`, `far`, `huge`.

I/O ACCESS

The program may access the H8 I/O system using the following data types:

`sfr, sfrp.`

NON-VOLATILE RAM

Variables may be placed in non-volatile RAM by using the following data type modifier:

`no_init.`

FUNCTION POINTERS

Function pointers are `near_func` in the small memory model and `far_func` in the large memory model. These defaults can be overridden by use of the `tiny_func` modifier:

`far_func, near_func, tiny_func.`

INTERRUPT ROUTINES

Interrupt handlers and non-interruptable routines may be written in C using the following keywords:

`interrupt, monitor.`

#PRAGMA DIRECTIVE SUMMARY

`#pragma` directives provide control of extension features while remaining within the standard language syntax.

Note that `#pragma` directives are available regardless of the `-e` option.

The following categories of `#pragma` functions are available:

BITFIELD ORIENTATION

`#pragma bitfield=default`
`#pragma bitfield=reversed`

CODE SEGMENT

`#pragma codeseg(seg-name)`

EXTENSION CONTROL

```
#pragma language=default
#pragma language=extended
```

FUNCTION ATTRIBUTE

```
#pragma function=default
#pragma function=interrupt
#pragma function=intrinsic
#pragma function=monitor
#pragma function=tiny_func
```

MEMORY USAGE

```
#pragma memory=constseg(seg-name)[:type]
#pragma memory=dataseg(seg-name)[:type]
#pragma memory=default
#pragma memory=far
#pragma memory=huge
#pragma memory=near
#pragma memory=no_init
#pragma memory=tiny
```

WARNING MESSAGE CONTROL

```
#pragma warnings=default
#pragma warnings=off
#pragma warnings=on
```

**PREDEFINED SYMBOLS
SUMMARY**

Predefined symbols allow inspection of the compile-time environment.

<i>Function</i>	<i>Description</i>
__DATE__	Current date in Mmm dd yyyy format.
__FILE__	Current source filename.
__IAR_SYSTEMS_ICC	IAR C compiler identifier.
__LINE__	Current source line number.
__STDC__	ANSI C compiler identifier.

<i>Function</i>	<i>Description</i>
<code>__TID__</code>	Target identifier.
<code>__TIME__</code>	Current time in hh:mm:ss format.
<code>__VER__</code>	Returns the version number as an <code>int</code> .

INTRINSIC FUNCTION SUMMARY

Intrinsic functions allow very low-level control of the H8 microprocessor. To use them in a C application, include the header file `inh8.h`. The intrinsic functions compile into in-line code, either a single instruction or a short sequence of instructions.

For details concerning the effects of the intrinsic functions, see the manufacturer's documentation of the H8 processor.

GENERAL INTRINSICS

The following intrinsics are available for all processor groups:

<i>Intrinsic</i>	<i>Description</i>
<code>_args\$</code>	Returns an array of the parameters to a function.
<code>_argt\$</code>	Returns the type of the parameter.
<code>and_ccr</code>	ANDs to the CCR register.
<code>dadd</code>	Performs decimal addition.
<code>disable_max_time</code>	Sets the maximum interrupt disable time.
<code>do_byte_eepmov (eepmov)</code>	Copies a sequence of bytes.
<code>do_word_eepmov (eepmov)</code>	Copies a sequence of words.
<code>dsub</code>	Performs decimal subtraction.
<code>func_stack_base</code>	Returns the function stack base address.
<code>get_imask_ccr</code>	Returns the interrupt mask of the condition code register.
<code>no_operation</code>	Executes the NOP instruction.
<code>or_ccr</code>	ORs to the CCR register.

<i>Intrinsic</i>	<i>Description</i>
ovfaddc, ovfaddw, ovfaddl	Adds 1-byte, 2-byte, or 4-byte data with overflow check.
ovfnegc, ovfnegw, ovfnegl	Negates 1-byte, 2-byte, or 4-byte data with overflow check.
ovfshalc, ovfshalw, ovfshall	Arithmetically shifts 1-byte, 2-byte, or 4-byte data with overflow check
ovfsubc, ovfsubw, ovfsubl	Subtracts 1-byte, 2-byte, or 4-byte data with overflow check.
read_ccr (get_ccr)	Reads the CCR register.
rotlc, rotlw, rotll	Rotates 1-byte, 2-byte, or 4-byte data to the left.
rotrc, rotrw, rotrl	Rotates 1-byte, 2-byte, or 4-byte data to the right.
set_imask_ccr	Sets the interrupt mask of the condition code register.
set_interrupt_mask	Sets the interrupt priority level.
sleep	Executes the SLEEP instruction.
tas	Executes the TAS instruction.
trapa	Executes the TRAPA instruction.
write_ccr (set_ccr)	Writes to the CCR register.
xor_ccr	Exclusive-ORs to the CCR register.

H8S/2600 INTRINSIC FUNCTIONS

The following additional intrinsic functions are available for the H8S/2600 processor group, selected using the -v2 command line option:

<i>Intrinsic</i>	<i>Description</i>
and_exr	ANDs to the EXR register.
get_imask_exr	Returns the interrupt mask of the extend register.
mac	Performs multiply and accumulate.
macl	Performs multiply and accumulate logical.
or_exr	ORs to the EXR register.
read_exr (get_exr)	Reads the EXR register.
read_hi_mac	Reads MACH.
read_mac	Reads MACL.
repeat_mac	Inserts a loop with the MAC instruction.
set_imask_exr	Sets the interrupt mask of the extend register.
single_mac	Performs a single MAC instruction.
write_exr (set_exr)	Writes to the EXR register.
write_ext_mac	Writes to MACH and MACL.
write_mac	Clears the MAC and writes to MACL.
xor_exr	XORs to the EXR register.

OTHER EXTENSIONS**\$ CHARACTER**

The character \$ has been added to the set of valid characters in identifiers for compatibility with DEC/VMS C.

USE OF SIZEOF AT COMPILE TIME

The ANSI-specified restriction that the sizeof operator cannot be used in #if and #elif expressions has been eliminated.

EXTENDED KEYWORD REFERENCE

This chapter describes the extended keywords in alphabetical order.

The following general parameters are used in several of the definitions:

<i>Parameter</i>	<i>What it means</i>
<i>storage-class</i>	Denotes an optional keyword <code>extern</code> or <code>static</code> .
<i>declarator</i>	Denotes a standard C variable or function declarator.

bit

Declares a bit variable.

SYNTAX – RELOCATABLE ADDRESS

bit identifier

SYNTAX – FIXED ADDRESS

bit identifier = constant-expression.bit-selector

SYNTAX – SFR

bit identifier = sfr-identifier.bit-selector

DESCRIPTION

The `bit` variable is a variable whose storage is a single bit. It may have values 0 and 1 only. Bit variables should not be confused with the C standard bitfields.

A bit variable can be one of three kinds:

<i>Bit variable type</i>	<i>Description</i>
Relocatable address	-v0 option: the variable is one bit of an ordinary relocatable variable in the tiny address range 0xFFFFFFFF00 to 0xFFFFFFFF. -v2 and -v3 options: the variable is one bit of an ordinary relocatable variable in the near address range 0xFFFF8000 to 0x00007FFF.
Fixed address	Must be in the bit addressable area 0x00000000 to 0xFFFFFFFF or 0xF0000000 to 0xFFFFFFFF.
sfr	The variable is one bit of an sfr variable.

far

Storage and pointer modifier.

SYNTAX

```
storage-class far declarator  
storage-class far * declarator
```

DESCRIPTION

The *far* modifier can only be specified in the large memory model, where it is the default. It allows you to specify *far* addressing when you have overridden the default with a `#pragma` directive.

EXAMPLES

The following large memory model program makes the default addressing mode huge and then specifies that *buffer* is *far*:

```
#pragma memory=huge  
int i; /* huge variable */  
far char buffer[1000]; /* large buffer in far memory */
```

The program needs a pointer into the *far* buffer:

```
char far *buffer_pointer;
```

Here the *far* keyword immediately before *** denotes a pointer of type *far*.

However, the following statement declares `buffer1` to be in far memory, while `buffer2` is placed in the default data area:

```
char far buffer1[1000], buffer2[1000];
```

far_func

Function or function pointer modifier.

SYNTAX

```
far_func function-declarator
storage-class far_func * declarator
```

DESCRIPTION

The `far_func` modifier specifies that the `far_func` calling mechanism is to be used for the function, or that a pointer points to a function declared with `far_func`.

In the large memory model the compiler uses the `far_func` mechanism by default, so you do not normally need to use the `far_func` modifier. It is provided so that if you use the `#pragma` directive to change the default mechanism, you can specify the `far_func` mechanism for an individual declaration.

In the small model, the `far_func` mechanism is not available.

EXAMPLE

```
#pragma function=tiny_func
/* Set default to tiny_func */

far_func void func2(void)          /* declare far_func
                                   function */
{
    ...
}
void (far_func * pi)()=func2;     /* pointer to above */
```

huge

Storage and pointer modifier.

SYNTAX

```
storage-class huge declarator  
storage-class huge * declarator
```

DESCRIPTION

The huge modifier can only be specified in the large memory model to override the default far addressing.

The huge modifier allows you to declare a data object in the huge segment with no restriction on size, or to specify that a pointer is to point to a data object in a huge segment. This lets you use additional memory for storing very large data objects.

EXAMPLES

The program defines a buffer using huge addressing:

```
int i;                               /* default far variable */  
huge char buffer[0x20000]; /* buffer larger than far  
                             capacity */
```

interrupt

Declare interrupt function.

SYNTAX

```
storage-class interrupt function-declarator  
storage-class interrupt [vector] function-declarator  
storage-class interrupt ccr_mask [mask] function-declarator  
storage-class interrupt [vector] ccr_mask [mask]  
function-declarator
```

PARAMETERS

function-declarator A function declarator.

[*vector*] A square-bracketed constant expression yielding the vector address.

[*mask*] A square-bracketed constant expression yielding the CCR mask.

DESCRIPTION

The `interrupt` keyword declares a function that is called upon a processor interrupt.

If the vector is a TRAPA vector, the interrupt function is allowed both to have parameters and return values. This is a software interrupt. The compiler will generate TRAPA instructions where these functions are called.

For other vectors the function must be void and have no arguments.

If a vector is specified, the address of the function is inserted in that vector. If no vector is specified, the user must provide an appropriate entry in the vector table (preferably placed in the `startup` module) for the interrupt function.

If a `ccr_mask` is specified, the first instruction of the function is manipulating the `ccr` register according to the specified mask. The low byte of the mask specifies the value which should be used. The high byte of the mask specifies the operation which should be used. 0 is `ldc`, 1 is `andc`, 2 is `orc`, and 3 is `xorc`. If no vector is specified, no manipulation of the `ccr` is done.

The run-time interrupt handler takes care of saving and restoring processor registers, and returning via the RTE instruction.

The compiler disallows calls to non-TRAPA interrupt functions from the program itself. It does allow interrupt function addresses to be passed to function pointers which do not have the `interrupt` attribute. This is useful for installing interrupt handlers in conjunction with operating systems.

EXAMPLES

Several include files are provided that define specific interrupt functions; see *Run-time library*, page 63. To use a predefined interrupt define it with a statement:

```
interrupt void name(void)
{ }
```

where *name* is the name of the interrupt function.

```
interrupt [0x4E] void SIO_interrupt(void)
{
    my_char=SBUF_RX;
    P1_REG=my_char;
}
```

monitor

Makes function atomic.

SYNTAX

storage-class **monitor** *function-declarator*

DESCRIPTION

The **monitor** keyword causes interrupts to be disabled during execution of the function. This allows atomic operations to be performed, such as operations on semaphores that control access to resources by multiple processes.

A function declared with **monitor** is equivalent to a normal function in all other respects.

EXAMPLES

The example below disables interrupts while the flag is tested. If the flag is not set, it is set. Interrupts are set to their previous state when the functions exits.

```
char printer_free;                /* printer-free
                                   semaphore */
monitor int got_flag(char *flag) /* With no danger of
                                   interruption ... */
{
    if (!*flag)                   /* test if available */
    {
        return (*flag = 1);      /* yes - take */
    }
    return (0);                   /* no - do not take */
}
void f(void)
{
```

```

    if (got_flag(&printer_free))    /* act only if
                                     printer is free */
        .... action code ....
}

```

near

Storage and pointer modifier.

SYNTAX

```

storage-class near declarator
storage-class near * declarator

```

DESCRIPTION

In the large memory model, the compiler normally places data objects in the far segment, accessing them by 32-bit addressing, and also allocates space for a far address in pointers to such data objects.

The near modifier allows you to place a data object in the near segment, where it is accessed by a more efficient 16-bit addressing mode, or to specify that a pointer is to point to a data object in a near segment. This lets you place frequently-accessed variables so they will be accessed more efficiently, and so that pointers to them will occupy 16 rather than 32 bits.

EXAMPLE

```

int i;                /* variable in default area */
near int in;         /* variable in near area */
int * pi;            /* pointer to variable in default
                    area */
near int * pi;       /* pointer to variable in near area
                    (pointer itself in default area) */
near int near * pi; /* pointer to variable in near area
                    (pointer itself also in near area) */

```

See also the examples for *far*, page 164.

near_func

Function or function pointer modifier.

SYNTAX

```
near_func function-declarator  
storage-class near_func * declarator
```

DESCRIPTION

The `near_func` modifier can only be specified in the small memory model, where it is the default. It allows you to define a `near_func` function when you have overridden the default with a `#pragma` directive.

EXAMPLE

```
#pragma function=tiny_func  
/* Set default to tiny_func */  
  
near_func void func2(void)      /* declare near_func  
                                function */  
{  
    ...  
}  
void (near_func * pi)()=func2; /* pointer to above */
```

no_init

Type modifier for non-volatile variables.

SYNTAX

```
storage-class no_init declarator
```

DESCRIPTION

By default, the compiler places variables in main, volatile RAM and initializes them on start-up. The `no_init` type modifier causes the compiler to place the variable in non-volatile RAM (or EEPROM) and not to initialize it on start-up.

`no_init` variable declarations may not include initializers.

If non-volatile memory is used, it is essential for the program to be linked to refer to the non-volatile RAM area. For details, see *Non-volatile RAM*, page 64.

EXAMPLES

The examples below show valid and invalid uses of the `no_init` modifier.

```
no_init int settings[50]; /* array of non-volatile
                          settings */
no_init far i ;          /* conflicting type
                          modifiers - invalid */
no_init int i = 1 ;     /* initializer included -
                          invalid */
```

sfr

Declare object of one-byte I/O data type.

SYNTAX

```
sfr identifier = constant-expression
```

DESCRIPTION

`sfr` denotes an I/O register which:

- ◆ Is equivalent to `unsigned char`.
- ◆ Can only be directly addressable; ie the `&` operator cannot be used.

The value of an `sfr` variable is the contents of the SFR register at the address *constant-expression*. All operators that apply to integral types except the unary `&` (address) operator may be applied to `sfr` variables.

In expressions, `sfr` variables may also be appended by a period followed by a bit-selector provided they lie in the bit area `0x00000000` to `0x0FFFFFFF` or `0xF0000000` to `0xFFFFFFFF`.

EXAMPLES

```
sfr P2 = 0xFFFFFFFF80;          /* Defines P2 */
void func()
```

```

{
    P2 = 4;                /* Set entire variable P2
                           = 00000100 */
    P2.2 = 1;             /* Only affects one bit
                           P2 = XXXXX1XX*/
    if (P2 & 4) printf("ON"); /* Read entire P2 and
                               mask bit 2 */
    if (P2.2) printf("ON"); /* Same but does bit
                               access only */
}

```

sfrp

Declare object of two-byte I/O data type.

SYNTAX

sfrp identifier = constant-expression

DESCRIPTION

sfrp denotes an I/O register which:

- ◆ Is equivalent to unsigned short.
- ◆ Can only be directly addressable; ie the & operator cannot be used.

The value of an sfrp variable is the contents of the SFR register at the address *constant-expression*. All operators that apply to integral types except the unary & (address) operator may be applied to sfrp variables.

In expressions, sfrp variables may be appended by a period followed by a bit-selector provided they lie in the bit area 0x00000000 to 0x0FFFFFFF or 0xF0000000 to 0xFFFFFFFF.

EXAMPLES

```

sfrp    P3CR = 0xFFFFFFFF90; /* Defines P3CR */
void func(void)
{
    P3CR = 0x400;             /* Set entire variable
                              P3CR = 00000100 00000000 */
    P3CR.10 = 1;            /* Only affects one bit
                              P3CR = xxxxx1xx xxxxxxxx */
}

```

```

    if (P3CR & 4) printf("ON"); /* read entire P3CR and
                                mask bit 10 */
    if (P3CR.10) printf("ON"); /* Same but does bit access
                                only */
}

```

tiny

Storage modifier.

SYNTAX*storage-class tiny declarator***DESCRIPTION**

The `tiny` modifier causes a data object to be placed in the `tiny` segment, `0xFFFFFFFF00` to `0xFFFFFFFFFF`, so it is accessed with the more-efficient 8-bit addressing mode. This lets you place frequently-used objects so they will be accessed most efficiently.

EXAMPLE

```

int i; /* variable in near segment */
tiny int is; /* variable in tiny segment */

```

tiny_func

Function modifier.

SYNTAX*tiny_func function-declarator***DESCRIPTION**

The `tiny_func` modifier causes the `tiny_func` calling mechanism to be used, even when this is not the default mechanism for the selected memory model.

`tiny_func` functions are called indirectly via an exception vector.

EXAMPLES

The following example shows a function `myfun` declared `tiny_func`:

```
#pragma language=extended
tiny_func void myfun()
```

#PRAGMA DIRECTIVE REFERENCE

This chapter describes the #pragma directives in alphabetical order.

bitfields = default

Restores default order of storage of bitfields.

SYNTAX

```
#pragma bitfields = default
```

DESCRIPTION

Causes the compiler to allocate bitfields in its normal order. See `bitfields=reversed`.

bitfields = reversed

Reverses order of storage of bitfields.

SYNTAX

```
#pragma bitfields=reversed
```

DESCRIPTION

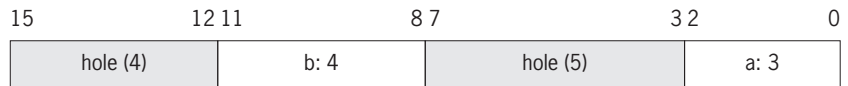
Causes the compiler to allocate bitfields starting at the most significant bit of the field, instead of at the least significant bit. The ANSI standard allows the storage order to be implementation dependent, so you can use this keyword to avoid portability problems.

EXAMPLES

The default layout of the following structure in memory is given in the diagram below:

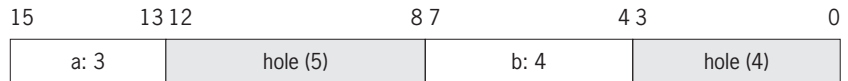
```
struct  
{  
    short a:3;    /* a is 3 bits */  
    short :5;    /* this reserves a hole of 5 bits */
```

```
short b:4;    /* b is 4 bits */
} bits;      /* bits is 16 bits */
```



For comparison, the following structure has the layout shown in the diagram below:

```
#pragma bitfields=reversed
struct
{
short a:3;    /* a is 3 bits */
short :5;    /* this reserves a hole of 5 bits */
short b:4;    /* b is 4 bits */
} bits;      /* bits is 16 bits */
```



codeseg

Sets the code segment name.

SYNTAX

```
#pragma codeseg(seg_name)
```

where *seg_name* specifies the segment name, which must not conflict with data segments.

DESCRIPTION

This directive places subsequent code in the named segment and is equivalent to using the -R option.

EXAMPLES

The following example defines the code segment as ROM:

```
#pragma codeseg(ROM)
```

function = default

Restores function definitions to the default type.

SYNTAX

```
#pragma function=default
```

DESCRIPTION

Returns function definitions to `near_func` or `far_func`, as set by the selected memory model.

EXAMPLES

The example below specifies that an external function `f1` can be called as a `tiny_func` function, while `f3` is the default type (`near_func` or `far_func` depending on the compiler options).

```
#pragma function=tiny_func
extern void f1();           /* Identical to extern
                           tiny_func void f1() */

#pragma function=default
extern int f3();           /* Default function type */
```

function = interrupt

Makes function definitions interrupt.

SYNTAX

```
#pragma function=interrupt
```

DESCRIPTION

This directive makes subsequent function definitions of `interrupt` type. It is an alternative to the function attribute `interrupt`.

Note that `#pragma function=interrupt` does not offer a vector option.

EXAMPLES

The example below shows an `interrupt` function `process_int` (the address of this function must be placed into the `INTVEC` table).

```
#pragma function=interrupt
void process_int()        /* an interrupt function */
```

function = intrinsic

```
{  
    ...  
}  
#pragma function=default
```

function = intrinsic

Replaces a function by an in-line code sequence.

SYNTAX

```
#pragma function=intrinsic(n)
```

PARAMETERS

n Determines when intrinsic functions are produced as in-line code:

<i>n</i>	<i>Condition</i>
----------	------------------

0	Controlled by the C compiler -s option.
---	---

1	Always on.
---	------------

2	Controlled by the C compiler -s or -z options.
---	--

3	Controlled by the C compiler -s option.
---	---

See *Optimize for speed (-s)*, page 42, and *Optimize for size (-z)*, page 44.

DESCRIPTION

Calls to certain C library functions can be replaced by an in-line sequence performed by the code-generator. This can change the characteristics of a function which may be undesirable. Therefore this optimization can be controlled by:

- ◆ A `#pragma` making the parser recognize intrinsic functions.
- ◆ An ANSI declaration found before the call.

If the call is rewritten into in-line the code looks like any other operator rather than a function call. That is because there is no need for the administration often required for function calls.

See also the chapter *Intrinsic function reference*.

EXAMPLE

The code below is from the include file `string.h`:

```
#if __TID__ & 0x8000          /* This processor knows
                               intrinsics */
#pragma function=intrinsic(n) /* "n" = 0 (see below) */
#endif
/* And now the declarations */
extern char *strcpy (char *, const char *);
extern int strlen (const char *);
#if __TID__ & 0x8000
#pragma function=default     /* Back to normal */
#endif
```

The high bit in `__TID__` indicates that the compiler supports this feature.

When an intrinsic function is found no external declaration is generated. However, as ANSI requires that it should also be possible to specify standard functions explicitly, there must also be a 'real' function in a library as well:

```
extern char *strcpy(char *, const char *);
                               /* Explicit declaration
                               */
char arr[80];
main()
{
    strcpy(arr,"hey");          /* Calls "real" function
                               */
}
```

Also, as it is possible to refer to functions indirectly, there will be external declarations if such are performed on intrinsic functions. The direct call though will be intrinsic.

```
#include <string.h>
char (*fp)(char *, const char *) = strcpy;
                               /* Indirect refer to lib */
char arr[80];
main()

{
```

function = monitor

```
strcpy(arr,"hey"); /* Intrinsic call/code */
}
```

When an intrinsic function definition (body) is found it will be treated as a standard function but calls to it will still be intrinsic. This can be used to create a minimal (and fast) ANSI-compatible library:

```
#include <string.h>
char *strcpy(char * d, const char *s)
    /* "real" function entry */
{
    return (strcpy(d,s)); /* Intrinsic call/code */
}
```

function = monitor

Makes function definitions atomic (non-interruptable).

SYNTAX

```
#pragma function=monitor
```

DESCRIPTION

Makes subsequent function definitions of monitor type. It is an alternative to the function attribute monitor.

EXAMPLES

The function f2 below will execute with interrupts temporarily disabled.

```
#pragma function=monitor
void f2() /* Will make f2 a monitor function */
{
    ...
}
#pragma function=default
```

function = tiny_func

Make subsequent function definitions default to the corresponding type.

SYNTAX

```
#pragma function=tiny_func
```

DESCRIPTION

These directives are alternatives to the function attributes.

EXAMPLE

```
#pragma function=tiny_func
void f2()                /* Will make f2 a tiny_func
                          function */
{
}
```

Make subsequent function definitions default to the corresponding type.

language = default

Restores availability of extended keywords to default.

SYNTAX

```
#pragma language=default
```

DESCRIPTION

Returns extended keyword availability to the default set by the C compiler `-e` option. See `language=extended`.

EXAMPLES

See the example `language=extended` below.

language = extended

Makes extended keywords available.

SYNTAX

```
#pragma language=extended
```

DESCRIPTION

Makes the extended keywords available regardless of the state of the C compiler `-e` option; see *Language extensions (-e)*, page 36.

EXAMPLE

In the example below, the `tiny` extended language modifier is enabled for the definition of the variable `ccount`. The variable `mycount` is defined in the standard way.

```
#pragma language=extended
tiny int ccount;          /* use single-byte addressing*/
#pragma language=default
int mycount;
```

memory = constseg

Directs constants to the named segment by default.

SYNTAX

```
#pragma memory=constseg(seg_name)[ : type]
```

DESCRIPTION

Directs constants to the named segment by default. It is an alternative to the memory attribute keywords. The default may be overridden by the memory attributes.

The segment must not be one of the compiler's reserved segment names.

The optional argument *type* can be used to specify the storage type, and can be one of:

`tiny`, `near`, `far`, or `huge`.

If omitted, constants will be placed in the default memory type.

EXAMPLE

The example below places the constant array `arr` into the ROM segment `TABLE`.

```
#pragma memory=constseg(TABLE)
char ar[] = {6, 9, 2, -5, 0};
#pragma memory = default
```

If another module accesses the array it must use an equivalent declaration:

```
#pragma memory=constseg(TABLE)
extern char * arr;
```


memory = dataseg

Directs variables to the named segment by default.

SYNTAX

```
#pragma memory=dataseg(seg_name)[ : type]
```

DESCRIPTION

Directs variables to the named segment by default. The default may be overridden by the memory attributes.

The optional argument *type* can be used to specify the storage type, and can be one of:

tiny, *near*, *far*, or *huge*.

If omitted, variables will be placed in the default memory type.

No initial values may be supplied in the variable definitions. Up to 10 different alternate data segments can be defined in any given module. You can switch to any previously defined data segment name at any point in the program.

EXAMPLE

The example below places three variables into the read/write area called USART.

```
#pragma memory=dataseg(USART)
char USART_data;           /* offset 0 */
char USART_control;       /* offset 1 */
int USART_rate;           /* offset 2, 3 */
#pragma memory = default
```

If another module wishes to access these symbols, the equivalent extern declaration should be used:

```
#pragma memory=dataseg(USART)
extern char USART_data;
```

memory = default

memory = default

Restores memory allocation of objects to the default area.

SYNTAX

```
#pragma memory=default
```

DESCRIPTION

Restores memory allocation of objects to the default area, as specified by the memory model in use.

memory = far

Directs variables to the far segment by default.

SYNTAX

```
#pragma memory=far
```

DESCRIPTION

Directs variables to the far area by default. This is only valid in the large memory model, where it is the default.

EXAMPLE

The example places the variable `buffer` into huge memory. It then restores far addressing for variable `i`:

```
#pragma memory=huge
int buffer[1000];           /* Buffer in huge memory */
#pragma memory=far
int i;                       /* Default far memory type */
```

memory = huge

Directs variables to the huge segment by default.

SYNTAX

```
#pragma memory=huge
```

DESCRIPTION

Directs variables to the huge area by default. This is only valid in the large memory model. The default can be overridden by the memory attribute.

EXAMPLE

The example places the variables `buffer` and `d` into huge memory. The `no_init` attribute of `strings` forces it into `no_init` memory.

```
#pragma memory=huge
int buffer[1000];           /* Buffer in huge memory
                             */
extern double d;           /* Variable in huge
                             memory */
no_init char *strings[5];  /* Overrides to no_init
                             memory */

#pragma memory=default
inti i;                    /* Default memory type */
```

memory = near

Directs variables to the near segment by default.

SYNTAX

```
#pragma memory=near
```

DESCRIPTION

Directs variables to the near area by default. The default can be overridden by the `memory` attribute.

EXAMPLE

The example places the variables `buffer` and `d` into near memory. The `no_init` attribute of `strings` forces it into `no_init` memory.

```
#pragma memory=near
int buffer[1000];           /* Buffer in near memory */
extern double d;           /* Variable in near
                             memory */
no_init char *strings[5];  /* Overrides to no_init
                             memory */

#pragma memory=default
inti i;                    /* Default memory type */
```

memory = no_init

Directs variables to the NO_INIT segment by default.

SYNTAX

```
#pragma memory=no_init
```

DESCRIPTION

Directs variables to the NO_INIT segment, so that they will not be initialized and will reside in non-volatile RAM. It is an alternative to the memory attribute `no_init`. The default may be overridden by the memory attributes.

The NO_INIT segment must be linked to coincide with the physical address of non-volatile RAM; see the chapter *Configuration* for details.

EXAMPLES

The example below places the variable `buffer` into non-initialized memory. Variables `i` and `j` are placed into the DATA area.

```
#pragma memory=no_init
char buffer[1000];    /* in uninitialized memory */
#pragma memory=default
int i,j;              /* default memory type */
```

Note that a non-default memory `#pragma` will generate error messages if function declarators are encountered. Local variables and parameters cannot reside in any other segment than their default segment, the stack.

memory = tiny

Directs variables to the tiny segment by default.

SYNTAX

```
#pragma memory=tiny
```

DESCRIPTION

Directs variables to the tiny area by default. The default can be overridden by the memory attribute.

warnings = default

Restores compiler warning output to default state

SYNTAX

```
#pragma warnings=default
```

DESCRIPTION

Returns the output of compiler warning messages to the default set by the C compiler `-w` option. See `#pragma warnings=on` and `#pragma warnings=off`.

warnings = off

Turns off output of compiler warnings.

SYNTAX

```
#pragma warnings=off
```

DESCRIPTION

Disables output of compiler warning messages regardless of the state of the C compiler `-w` option; see *Disable warnings (-w)*, page 43.

warnings = on

Turns on output of compiler warnings.

SYNTAX

```
#pragma warnings=on
```

DESCRIPTION

Enables output of compiler warning messages regardless of the state of the C compiler `-w` option; see *Disable warnings (-w)*, page 43.

warnings = on

PREDEFINED SYMBOLS REFERENCE

This chapter gives reference information about the symbols predefined by the compiler.

__DATE__

Current date.

SYNTAX

`__DATE__`

DESCRIPTION

The date of compilation is returned in the form Mmm dd yyyy.

__FILE__

Current source filename.

SYNTAX

`__FILE__`

DESCRIPTION

The name of the file currently being compiled is returned.

__IAR_SYSTEMS_ICC

IAR C compiler identifier.

SYNTAX

`__IAR_SYSTEMS_ICC`

DESCRIPTION

The number 1 is returned. This symbol can be tested with `#ifdef` to detect being compiled by an IAR Systems C Compiler.

__LINE__

__LINE__

Current source line number.

SYNTAX

__LINE__

DESCRIPTION

The current line number of the file currently being compiled is returned.

__STDC__

IAR C compiler identifier.

SYNTAX

__STDC__

DESCRIPTION

The number 1 is returned. This symbol can be tested with `#ifdef` to detect being compiled by an ANSI C compiler.

__TID__

Target identifier.

SYNTAX

__TID__

DESCRIPTION

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

31	16	15	14	8	7	4	3	0
(not used)	Intrinsic support	Target_IDENT, unique to each target processor	-v option value, if supported	-m option value, if supported				

The `__TID__` value is constructed as:

```
(0x8000 | (t << 8) | (v << 4) | m)
```

You can extract the values as follows:

```
f = (__TID__) & 0x8000;
t = (__TID__ >> 8) & 0x7F;
v = (__TID__ >> 4) & 0xF;
m = __TID__ & 0x0F;
```

Note that there are two underscores at each end of the macro name.

To find the value of `Target_IDENT` for the current compiler, execute:

```
printf("%ld", (__TID__ >> 8) & 0x7F)
```

For an example of the use of `__TID__`, see the file `stdarg.h`.

The highest bit `0x8000` is set in the H8 C Compiler to indicate that the compiler recognizes intrinsic functions. This may affect how you write header files.

__TIME__

Current time.

SYNTAX

```
__TIME__
```

DESCRIPTION

The time of compilation is returned in the form `hh:mm:ss`.

__VER__

Returns the compiler version number.

SYNTAX

```
__VER__
```

DESCRIPTION

The version number of the compiler is returned as an `int`.

EXAMPLE

The example below prints a message for version 3.34.

```
#if __VER__ == 334
#message "Compiler version 3.34"
#endif
```

INTRINSIC FUNCTION REFERENCE

This chapter gives reference information about the intrinsic functions. To use the intrinsic functions include the header file `inh8.h`.

In addition to the IAR H8 intrinsics, the H8 C Compiler also supports the intrinsic functions provided by the Hitachi H8 C Compiler and, where appropriate, these are listed in brackets after the equivalent IAR function.

Certain intrinsics, marked `-v2` only, are only available for the H8S/2600 processor group.

`_args$`

Returns an array of the parameters to a function.

SYNTAX

`_args$`

DESCRIPTION

`_args$` is a reserved word that returns a char array (`char *`) containing a list of descriptions of the formal parameters of the current function:

<i>Offset</i>	<i>Contents</i>
0	Parameter 1 type in <code>_argt\$</code> format.
1	Parameter 1 size in bytes.
2	Parameter 2 type in <code>_argt\$</code> format.
3	Parameter 2 size in bytes.
2n-2	Parameter n type in <code>_argt\$</code> format.
2n-1	Parameter n size in bytes.
2n	<code>\0</code>

Sizes greater than 127 are reported as 127.

`_argt$`

`_argt$` may be used only inside function definitions. For an example of the use of `_argt$`, see the file `stdarg.h`.

If a variable length (`varargs`) parameter list was specified then the parameter list is deemed to terminate at the final explicit parameter; you cannot easily determine the types or sizes of the optional parameters.

`_argt$`

Returns the type of the parameter.

SYNTAX

`_argt$(v)`

DESCRIPTION

The returned values and their corresponding meanings are shown in the following table.

<i>Value</i>	<i>Type</i>
1	unsigned char
2	char
3	unsigned short
4	short
5	unsigned int
6	int
7	unsigned long
8	long
9	float
10	double
11	long double
12	pointer/address
13	union
14	struct

EXAMPLE

The example below uses `_argt$` and tests for integer or long parameter types.

```
switch (_argt$(i))
{
    case 6:
```

```
    printf("int %d\n", i);
    break;
case 8:
    printf("long %ld\n", i);
    break;
default:
```

and_ccr

ANDs to the CCR register.

SYNTAX

```
void and_ccr(unsigned char mask)
```

DESCRIPTION

CCR &= *mask*. The function argument *mask* should be a constant.

and_exr

ANDs to the EXR register (-v2 only).

SYNTAX

```
void and_exr(unsigned char mask)
```

DESCRIPTION

EXR &= *mask*. The function argument *mask* should be a constant.

dadd

Performs decimal addition.

SYNTAX

```
void dadd (unsigned char size, char *ptr1, char *ptr2,
char *rst)
```

DESCRIPTION

Adds *size*-byte data stored in the area at *ptr1* to *size*-byte data stored in the area at *ptr2* and stores the result in the *size*-byte area at *rst*. The *size* must be a constant from 1 to 255.

disable_max_time

Sets the maximum interrupt disable time.

SYNTAX

```
void disable_max_time(unsigned long cycles)
```

DESCRIPTION

Informs the compiler if it is possible to use the `EEPMOV.B` instruction for moving blocks of memory. Since the `EEPMOV.B` instruction turns off the interrupts, it is essential to be sure that the interrupts are not turned off too long time. Note that the calculation is made for moving data within 2 cycle memory for the `-v0` processor group option, and 1 cycle memory for the `-v1` and `-v2` options.

do_byte_eepmov (eepmov)

Copies a sequence of bytes.

SYNTAX

```
void do_byte_eepmov(char *source, char *dest, unsigned  
char count)
```

DESCRIPTION

Copies a sequence of *count* bytes from the address specified by *source* to an EEPROM location specified by *dest*. The `EEPMOV.B` instruction is used.

do_word_eepmov (eepmov)

Copies a sequence of words.

SYNTAX

```
void do_word_eepmov(char *source, char *dest, unsigned  
char count)
```

DESCRIPTION

Copies a sequence of *count* words from the address specified by *source* to an EEPROM location specified by *dest*. The `EEPMOV.W` instruction is used.

dsub

Performs decimal subtraction.

SYNTAX

```
void dsub (unsigned char size, char *ptr1, char *ptr2,  
char *rst)
```

DESCRIPTION

Subtracts *size*-byte data stored in the area at *ptr1* to *size*-byte data stored in the area at *ptr2* and stores the result in the *size*-byte area at *rst*. The size must be a constant from 1 to 255.

func_stack_base

Returns the function stack base address.

SYNTAX

```
void *func_stack_base(void)
```

DESCRIPTION

Gives the value which the stack pointer SP had when the current function was entered. This means that the result of this intrinsic call points to the return address of the current function.

get_imask_ccr

Returns the interrupt mask of the condition code register.

SYNTAX

```
unsigned char get_imask_ccr(void)
```

DESCRIPTION

Returns the mask value (0 to 1) in the interrupt mask bit (1) of the condition code register (CCR).

get_imask_exr

Returns the interrupt mask of the extend register (-v2 only).

SYNTAX

```
unsigned char get_imask-exr(void)
```

DESCRIPTION

Returns the mask value (0 to 7) in the interrupt mask bits (I2 to I0) of the extend register (EXR).

mac

Performs multiply and accumulate (-v2 only).

SYNTAX

```
void mac(long val, int *ptr1, int *ptr2, unsigned long count)
```

DESCRIPTION

Sets *val* to the MAC register as the initial value, multiplies two bytes *ptr1* and *ptr2* with sign, adds the 4-byte result to the MAC register contents, and adds two to *ptr1* and *ptr2*. This operation is repeated for the number of times specified by *count*.

macl

Performs multiply and accumulate logical (-v2 only).

SYNTAX

```
void macl(long val, int *ptr1, int *ptr2, unsigned long count, unsigned long mask)
```

DESCRIPTION

As *mac*, but it logically ANDs *ptr2* with *mask* to use *ptr2* repeatedly.

no_operation

Executes the NOP instruction.

SYNTAX

```
void no_operation(void)
```

DESCRIPTION

Executes the NOP instruction.

or_ccr

ORs to the CCR register.

SYNTAX

```
void or_ccr(unsigned char mask)
```

DESCRIPTION

CCR |= *mask*. The function argument *mask* should be a constant.

or_exr

ORs to the EXR register (-v2 only).

SYNTAX

```
void or_exr(unsigned char mask)
```

DESCRIPTION

EXR |= *mask*. The function argument *mask* should be a constant.

**ovfaddc, ovfaddw,
ovfaddl**

Adds 1-byte, 2-byte, or 4-byte data with overflow check.

SYNTAX

```
int ovfaddc(char dst, char src, char *rst)
int ovfaddw(int dst, int src, int *rst)
int ovfaddl(long dst, long src, long *rst)
```

DESCRIPTION

Adds 1-byte, 2-byte, and 4-byte data *dst* and *src*, stores the results in the area specified by *rst* only when *rst* is not 0, returns 0 when the results do not overflow and a value other than 0 when they do overflow.

These functions can be used only in the conditional statements such as the *if*, *do*, *while*, and *for* statements.

ovfnegc, ovfnegw ovfnegl

Negates 1-byte, 2-byte, or 4-byte data with overflow check.

SYNTAX

```
int ovfnegc(char dst, char *rst)
int ovfnegw(int dst, int *rst)
int ovfnegl(long dst, long *rst)
```

DESCRIPTION

Calculates the 2's complements of 1-byte, 2-byte, and 4-byte data *dst*, stores the results in the area specified by *rst* only when *rst* is not 0, and returns 0 when the results do not overflow and a value other than 0 when they overflow.

These functions can be used only in the conditional statements such as the *if*, *do*, *while*, and *for* statements.

ovfshalc, ovfshalw ovfshall

Arithmetically shifts 1-byte, 2-byte, or 4-byte data with overflow check.

SYNTAX

```
int ovfshalc(char dst, char char *rst)
int ovfshalw(int dst, int *rst)
int ovfshall(long dst, long *rst)
```

DESCRIPTION

Arithmetically shifts 1-byte, 2-byte, and 4-byte data *dst* to the left by one bit, stores the results in the area specified by *rst* only when *rst* is not 0, and returns 0 when the results do not overflow and a value other than 0 when they overflow.

These functions can be used only in the conditional statements such as the `if`, `do`, `while`, and `for` statements.

ovfsubc, ovfsubw ovfsubl

Subtracts 1-byte, 2-byte, or 4-byte data with overflow check.

SYNTAX

```
int ovfsubc(char dst, char src, char *rst)
int ovfsubw(int dst, int src, int *rst)
int ovfsubl(long dst, long src, long *rst)
```

DESCRIPTION

Subtracts 1-byte, 2-byte, and 4-byte data *src* from *dst*, stores the results in the area specified by *rst* only when *rst* is not 0, and returns 0 when the results do not overflow and a value other than 0 when they overflow.

These functions can be used only in the conditional statements such as the `if`, `do`, `while`, and `for` statements.

read_ccr (get_ccr)

Reads the CCR register.

SYNTAX

```
unsigned char read_ccr(void)
```

DESCRIPTION

Reads the CCR (condition code) register.

read_exr (get_exr)

Reads the EXR register (-v2 only).

SYNTAX

```
unsigned char read_exr(void)
```

DESCRIPTION

Reads the EXR (extended) register.

read_hi_mac

Reads the MACH register (-v2 only).

SYNTAX

```
long read_hi_mac(void)
```

DESCRIPTION

Reads the MACH register.

read_mac

Reads the MACL register (-v2 only).

SYNTAX

```
long read_mac(void)
```

DESCRIPTION

Reads the MACL register.

repeat_mac

Inserts a loop with the MAC instruction (-v2 only).

SYNTAX

```
void repeat_mac(int *ptr1, int *ptr2, unsigned long  
count)
```

DESCRIPTION

Inserts a loop with the MAC instruction.

rotlc, rotlw, rotll

Rotates 1-byte, 2-byte, or 4-byte data to the left.

SYNTAX

```
char rotlc(int count, char data)  
int rotlw(int count, int data)  
long rotll(int count, long data)
```

DESCRIPTION

Rotates 1-byte, 2-byte, or 4-byte data to the left by *count* bits, and returns the result.

rotrc, rotrw, rotrl

Rotates 1-byte, 2-byte, or 4-byte data to the right.

SYNTAX

```
char rotrc(int count, char data)
int rotrw(int count, int data)
long rotrl(int count, long data)
```

DESCRIPTION

Rotates 1-byte, 2-byte, and 4-byte data to the right by *count* bits, and returns the result.

set_imask_ccr

Sets the interrupt mask of the condition code register.

SYNTAX

```
void set_imask_ccr(unsigned char mask)
```

DESCRIPTION

Sets the mask value (0 to 1) to the interrupt mask bit (1) of the condition code register (CCR).

set_imask_exr

Sets the interrupt mask of the extend register (-v2 only).

SYNTAX

```
void set_imask_exr(unsigned char mask)
```

DESCRIPTION

Sets the mask value (0 to 7) to the interrupt mask bits (I2 to I0) of the extend register (EXR).

set_interrupt_mask

Sets the interrupt priority level.

SYNTAX

```
void set_interrupt_mask(char mask)
```

DESCRIPTION

Sets the interrupt priority level. The parameter, which should be a constant, is interpreted as shown below:

<i>Argument</i>	<i>Action</i>
0	All interrupts enabled.
1	All interrupts enabled.
2	General interrupts disabled.
3	All interrupts except NMI disabled.

single_mac

Performs a single MAC instruction (-v2 only).

SYNTAX

```
void single_mac(int *ptr1, int *ptr2)
```

sleep

Executes the SLEEP instruction.

SYNTAX

```
void sleep(void)
```

DESCRIPTION

Executes the SLEEP instruction.

tas

Executes the TAS instruction.

SYNTAX

```
void tas(char *addr)
```

DESCRIPTION

Expanded to the test and set instruction, TAS.

trapa

Executes the TRAPA instruction.

SYNTAX

```
void trapa(unsigned int trap_no)
```

DESCRIPTION

Expanded to an unconditional trap instruction, TRAPA#*trap_no*.

write_ccr (set_ccr)

Writes to the CCR register.

SYNTAX

```
void write_ccr(unsigned char value)
```

DESCRIPTION

Changes the contents of the CCR register to value.

write_exr (set_exr)

Writes to the EXR register (-v2 only).

SYNTAX

```
void set_exr(unsigned char value)
```

DESCRIPTION

Writes to the EXR (extended) register.

write_ext_mac

Writes to MACH and MACL (-v2 only).

SYNTAX

```
void write_ext_mac(long hi_val, long lo_val)
```

DESCRIPTION

Puts *hi_val* into MACH and *lo_val* into MACL.

write_mac

Clears the MAC and writes to MACL (-v2 only).

SYNTAX

```
void write_mac(long val)
```

DESCRIPTION

Clears the MAC register using a CLRMAC instruction, and then puts *val* into MACL.

xor_ccr

Exclusive-ORs to the CCR register.

SYNTAX

```
void xor_ccr(unsigned char mask)
```

DESCRIPTION

CCR ^= *mask*. The function argument *mask* should be a constant.

xor_exr

XORs to the EXR register (-v2 only).

SYNTAX

```
void xor_exr(unsigned char value)
```

DESCRIPTION

EXR ^= *mask*. The function argument *mask* should be a constant.

ASSEMBLY LANGUAGE INTERFACE

The H8 C Compiler allows assembly language modules to be combined with compiled C modules. This is particularly used for small, time-critical routines that need to be written in assembly language and then called from a C main program. This chapter describes the interface between a C main program and assembly language routines.

CREATING A SHELL

The recommended method of creating an assembly language routine with the correct interface is to start with an assembly language source created by the C compiler. To this shell you can easily add the functional body of the routine.

The shell source needs only to declare the variables required and perform simple accesses to them, for example:

```
int k;
int foo(int i, int j)
{
    char c;
    i++;          /* Access to i */
    j++;          /* Access to j */
    c++;          /* Access to c */
    k++;          /* Access to k */
}
void f(void)
{
    foo(4,5);     /* Call to foo */
}
```

This program is then compiled as follows:

```
icch8 shell -A -q -L ↵
```

The `-A` option creates an assembly language output, the `-q` option includes the C source lines as assembler comments, and the `-L` option creates a listing.

The result is the assembler source shell.s37 containing the declarations, function call, function return, variable accesses, and a listing file shell.lst.

The following sections describe the interface in detail.

CALLING CONVENTION

The C compiler uses the run-time stack for administrating function calls. Space for parameters and auto variables is allocated on the stack. Although the first parameter has a location on the stack, it is normally transferred in registers:

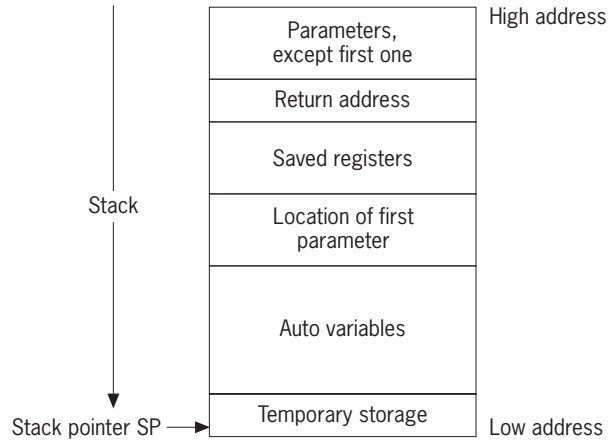
<i>Size of parameter</i>	<i>Location of parameter</i>
1 byte	R6L
2	R6
4	ER6

The only exception to this rule is when the first parameter is a `struct`, `union`, or an 8-byte `double`, in which case the parameter is pushed on the stack.

The remaining parameters, which are not transferred in registers, are pushed on the stack in reverse order, ie the last parameter is transferred first. Pushed parameters are removed by the caller after returning from the called function.

All registers are preserved across function calls except for ER5 and ER6 which are used as scratch registers. Each function is therefore responsible for saving and restoring any register it will use. The condition codes in CCR are not preserved, except for `interrupt` and `monitor` functions. Another exception to this rule are the registers that transfer the function return value.

To sum up, a stack frame has the following general layout in memory:



The return value is given in registers if possible, otherwise at the pointed-to location in the caller's own storage space:

<i>Size of return value</i>	<i>Location of return value</i>
1 byte	R6L
2	R6
4	ER6
More than 4 bytes, struct, union, or 8-byte double.	Pointed to by R6 or ER6 depending on memory model.

Note that `struct jim foo()` is converted internally to `void foo(struct jim *)` to allow the caller to specify where to store the result.

CALLING ASSEMBLY ROUTINES FROM C

An assembler routine that is to be called from C must:

- ◆ Conform to the calling convention described above.
- ◆ Have a PUBLIC entry-point label.
- ◆ Be declared as external before any call, to allow type checking and optional promotion of parameters, as in `extern int foo()` or `extern int foo(int i, int j)`.

LOCAL STORAGE ALLOCATION

If the routine needs local storage, it may allocate it in one or more of the following ways:

- ◆ On the hardware stack.
- ◆ In static workspace, provided of course that the routine is not required to be simultaneously re-usable (“re-entrant”).

Functions can always use ER5 to ER6 without saving them.

INTERRUPT FUNCTIONS

The calling convention cannot be used for interrupt functions since the interrupt may occur during the calling of a foreground function. Hence the requirements for interrupt function routine are different from those of a normal function routine, as follows:

- ◆ The routine must preserve all used registers.
- ◆ The routine must exit using RTE.
- ◆ The routine must treat all flags as undefined.

DEFINING INTERRUPT VECTORS

As an alternative to defining a C interrupt function in assembly language as described above, the user is free to assemble an interrupt routine and install it directly in the interrupt vector.

The interrupt vectors are located in the INTVEC segment.

SEGMENT REFERENCE

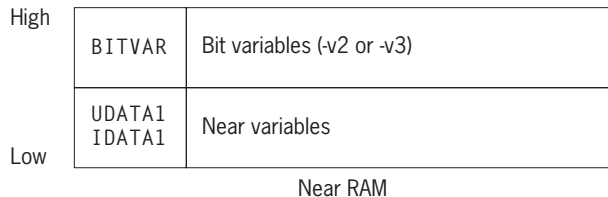
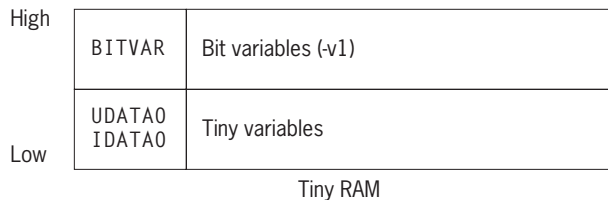
The H8 C Compiler places code and data into named segments which are referred to by XLINK. Details of the segments are required for programming assembly language modules, and are also useful when interpreting the assembly language output of the compiler.

This section provides an alphabetical list of the segments. For each segment, it shows:

- ◆ The name of the segment.
- ◆ A brief description of the contents.
- ◆ Whether the segment is read/write or read-only.
- ◆ Whether the segment may be accessed from the assembly language (assembly-accessible) or from the compiler only.
- ◆ A fuller description of the segment contents and use.

MEMORY MAP DIAGRAMS

The diagrams on the following pages show the H8 memory map and the allocation of segments within each memory area.



SEGMENT REFERENCE

High	TEMP	Auto variables when compiling with the -d option
	NO_INIT	Non-volatile variables
	CSTACK	Hardware stack
	ECSTR WCSTR	Writable string variables
	UDATA2 IDATA2	Far variables
Low	UDATA3 IDATA3	Huge variables

RAM

0xFF	FLIST IFLIST	tiny_func function table
0x00	INTVEC	Interrupt vector table

PROM 0x00 to 0xFF

High	CCSTR	String literal initializers
	CDATA0 CDATA1 CDATA2 CDATA3	Variable initializers
	CSTR	Constant string literals
	CONST	Constants
Low	CODE RCODE	Code and program code

PROM

BITVAR

Bit variables.

TYPE

Read-write.

DESCRIPTION

Assembler-accessible.

Holds bit variables and can also hold user-written relocatable bit-variables.

CCSTR

String literals.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible.

Holds C string literals. This segment is copied to ECSTR at startup. For more information refer to *Writable strings (-y)*, page 44. See also *CSTR*, page 215, and *ECSTR*, page 215.

**CDATA0, CDATA1,
CDATA2, CDATA3**

Initialization constants for tiny, near, far and huge data, respectively.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible.

CSTARTUP copies initialization values from this segment to the IDATA0 ... IDATA3 segments.

CODE

Code.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible.

Holds user program code and various library routines. Note that any assembly language routines called from C must meet the calling convention of the memory model in use. For more information see *Calling assembly routines from C*, page 210.

CONST

Constants.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible.

Used for storing const objects. Can be used in assembly language routines for declaring constant data.

CSTACK

Stack.

TYPE

Read/write.

DESCRIPTION

Assembly-accessible.

Holds the internal stack.

This segment and length is normally defined in the XLINK file by the command:

```
-Z(DATA)CSTACK + nn = start
```

where *nn* is the length and *start* is the location.

CSTR

String literals.

TYPE

Read only.

DESCRIPTION

Assembly-accessible.

Holds C string literals when the C compiler **Writable strings** (*-y*) option is not active (default). For more information see *Writable strings* (*-y*), page 44. See also *CCSTR*, page 213, and *ECSTR*, page 215.

ECSTR

Writable copies of string literals.

TYPE

Read/write.

DESCRIPTION

Assembly-accessible.

Holds C string literals. For more information refer to *Writable strings* (*-y*), page 44. See also *CCSTR*, page 213, and *CSTR*, page 215.

FLIST, IFLIST

Function list.

TYPE

Read-only.

DESCRIPTION

Assembler-accessible.

Holds a function table that is used to call functions by the `tiny_func` mechanism. A `tiny_func` reference is an 8-bit index into this table. The address to the actual function is contained in the `FLIST`/`IFLIST` entry. The `IFLIST` segment contains references to internal functions of the run-time library, while `FLIST` contains references to user-written `tiny_func` functions.

**IDATA0, IDATA1,
IDATA2, IDATA3**

Initialized static data for tiny, near, far and huge data, respectively.

TYPE

Read-write.

DESCRIPTION

Assembly-accessible.

Holds static variables in internal data memory that are automatically initialized from `CDATA0 ... CDATA3` in `cstartup.s37`. See also `CDATAN` above.

INTVEC

Interrupt vectors.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible.

Holds the interrupt vector table generated by the use of the `interrupt` extended keyword (which can also be used for user-written interrupt vector table entries).

NO_INIT

Non-volatile variables.

TYPE

Read/write.

DESCRIPTION

Assembly-accessible.

Holds variables to be placed in non-volatile memory. These will have been allocated by the compiler, declared `no_init` or created `no_init` by use of the memory `#pragma`, or created manually from assembly language source.

RCODE

Startup code.

TYPE

Read-only.

DESCRIPTION

Assembly-accessible.

Used for interrupt handlers and internal library functions.

TEMP

Autos.

TYPE

Read/write.

DESCRIPTION

Used for autos when compiling with the `-d` option.

**UDATA0, UDATA1,
UDATA2, UDATA3**

Uninitialized static data for tiny, near, far, and huge data respectively.

TYPE

Read/write.

DESCRIPTION

Assembly-accessible.

Holds variables in memory that are not explicitly initialized; these are implicitly initialized to all zero, which is performed by CSTARTUP.

WCSTR

Writable string literals.

TYPE

Read/write.

DESCRIPTION

Assembler-accessible.

Holds writable copies of C string literals when the compiler's -P option is inactive. See *Generate promable code (-P)*, page 59.

K&R AND ANSI C LANGUAGE DEFINITIONS

This chapter describes the differences between the K&R description of the C language and the ANSI standard.

INTRODUCTION

There are two major standard C language definitions:

- ◆ Kernighan & Richie, commonly abbreviated to K&R.

This is the original definition by the authors of the C language, and is described in their book *The C Programming Language*.

- ◆ ANSI.

The ANSI definition is a development of the original K&R definition. It adds facilities that enhance portability and parameter checking, and removes a small number of redundant keywords. The IAR Systems C Compiler follows the ANSI approved standard X3.159-1989.

Both standards are described in depth in the latest edition of *The C Programming Language* by Kernighan & Richie. This chapter summarizes the differences between the standards, and is particularly useful to programmers who are familiar with K&R C but would like to use the new ANSI facilities.

DEFINITIONS

ENTRY KEYWORD

In ANSI C the entry keyword is removed, so allowing entry to be a user-defined symbol.

CONST KEYWORD

ANSI C adds `const`, an attribute indicating that a declared object is unmodifiable and hence may be compiled into a read-only memory segment. For example:

```
const int i;           /* constant int */
const int *ip;        /* variable pointer to
                      constant int */
```

```
int *const ip;                /* constant pointer to
                             variable int */
typedef struct                /* define the struct
                             'cmd_entry' */
{
    char *command;
    void (*function)(void);
}
cmd_entry
const cmd_entry table[]=    /* declare a constant object
                             of type 'cmd_entry' */
{
    "help", do_help,
    "reset", do_reset,
    "quit", do_quit
};
```

VOLATILE KEYWORD

ANSI C adds `volatile`, an attribute indicating that the object may be modified by hardware and hence any access should not be removed by optimization.

SIGNED KEYWORD

ANSI C adds `signed`, an attribute indicating that an integer type is signed. It is the counterpart of `unsigned` and can be used before any integer type-specifier.

VOID KEYWORD

ANSI C adds `void`, a type-specifier that can be used to declare function return values, function parameters, and generic pointers. For example:

```
void f();                    /* a function without return
                             value */
type_spec f(void);          /* a function with no parameters
                             */
void *p;                    /* a generic pointer which can be
                             /* cast to any other pointer and
                             is assignment-compatible with any
                             pointer type */
```

ENUM KEYWORD

ANSI C adds enum, a keyword that conveniently defines successive named integer constants with successive values. For example:

```
enum {zero,one,two,step=6,seven,eight};
```

DATA TYPES

In ANSI C the complete set of basic data types is:

```
{unsigned | signed} char
{unsigned | signed} int
{unsigned | signed} short
{unsigned | signed} long
float
double
long double
*                /* Pointer */
```

FUNCTION DEFINITION PARAMETERS

In K&R C, function parameters are declared by conventional declaration statements before the body of the function. In ANSI C, each parameter in the parameter list is preceded by its type identifiers. For example:

<i>K&R</i>	<i>ANSI</i>
long int g(s) char * s;	long int g (char * s)
{	{

The arguments of ANSI-type functions are always type-checked. The IAR Systems C Compiler checks the arguments of K&R-type functions only if the **Global strict type check** (-g) option is used.

FUNCTION DECLARATIONS

In K&R C, function declarations do not include parameters. In ANSI C they do. For example:

<i>Type</i>	<i>Example</i>
K&R	<code>extern int f();</code>
ANSI (named form)	<code>extern int(long int val);</code>
ANSI (unnamed form)	<code>extern int(long int);</code>

In the K&R case, a call to the function via the declaration cannot have its parameter types checked, and if there is a parameter-type mismatch, the call will fail.

In the ANSI C case, the types of function arguments are checked against those of the parameters in the declaration. If necessary, a parameter of a function call is cast to the type of the parameter in the declaration, in the same way as an argument to an assignment operator might be. Parameter names are optional in the declaration.

ANSI also specifies that to denote a variable number of arguments, an ellipsis (three dots) is included as a final formal parameter.

If external or forward references to ANSI-type functions are used, a function declaration should appear before the call. It is unsafe to mix ANSI and K&R type declarations since they are not compatible for promoted parameters (char or float).

Note that in the IAR Systems C Compiler, the -g option will find all compatibility problems among function calls and declarations, including between modules.

HEXADECIMAL STRING CONSTANTS

ANSI allows hexadecimal constants denoted by backslash followed by x and any number of hexadecimal digits. For example:

```
#define Escape_C "\x1b\x43" /* Escape 'C' \0 */
```

\x43 represents ASCII C which, if included directly, would be interpreted as part of the hexadecimal constant.

STRUCTURE AND UNION ASSIGNMENTS

In K&R C, functions and the assignment operator may have arguments that are pointers to struct or union objects, but not struct or union objects themselves.

ANSI C allows functions and the assignment operator to have arguments that are struct or union objects, or pointers to them. Functions may also return structures or unions:

```
struct s a,b;                /* struct s declared earlier
                             */
struct s f(struct s parm); /* declare function
                             accepting and returning
                             structs */
a = f(b);                    /* call it */
```

To increase the usability of structures further, ANSI allows auto structures to be initialized.

SHARED VARIABLE OBJECTS

Various C compilers differ in their handling of variable objects shared among modules. The IAR Systems C Compiler uses the scheme called *Strict REF/DEF*, recommended in the ANSI supplementary document *Rationale For C*. It requires that all modules except one use the keyword *extern* before the variable declaration. For example:

<i>Module #1</i>	<i>Module #2</i>	<i>Module #3</i>
int i;	extern int i;	extern int i;
int j=4;	extern int j;	extern int j;

#elif

ANSI C's new *#elif* directive allows more compact nested else-if structures.

```
#elif expression
...
```

is equivalent to:

```
#else
#if expression
```

```
...  
#endif
```

#error

The `#error` directive is provided for use in conjunction with conditional compilation. When the `#error` directive is found, the compiler issues an error message and terminates.

DIAGNOSTICS

The diagnostic error and warning messages fall into six categories:

- ◆ Command line error messages.
- ◆ Compilation error messages.
- ◆ Compilation warning messages.
- ◆ Compilation fatal error messages.
- ◆ Compilation memory overflow message.
- ◆ Compilation internal error messages.

COMMAND LINE ERROR MESSAGES

Command line errors occur when the compiler finds a fault in the parameters given on the command line. In this case, the compiler issues a self-explanatory message.

COMPILATION ERROR MESSAGES

Compilation error messages are produced when the compiler has found a construct which clearly violates the C language rules, such that code cannot be produced.

IAR C compilers are more strict on compatibility issues than many other C compilers. In particular pointers and integers are considered as incompatible when not explicitly casted.

COMPILATION WARNING MESSAGES

Compilation warning messages are produced when the compiler finds a programming error or omission which is of concern but not so severe as to prevent the completion of compilation.

COMPILATION FATAL ERROR MESSAGES

Compilation fatal error messages are produced when the compiler has found a condition that not only prevents code generation, but which makes further processing of the source not meaningful. After the message has been issued, compilation terminates. Compilation fatal error messages are described in *Compilation error messages* in this chapter, and marked as fatal.

COMPILATION MEMORY OVERFLOW MESSAGE

When the compiler runs out of memory, it issues the special message:

```
* * * C O M P I L E R   O U T   O F   M E M O R Y * * *  
      Dynamic memory used: nnnnnn bytes
```

If this error occurs, the cure is either to add system memory or to split source files into smaller modules. Also note that the following options cause the compiler to use more memory (but not -rn):

<i>Option</i>	<i>Command line</i>
Insert mnemonics.	-q
Cross-reference.	-x
Assembly output to prefixed filename.	-A
Generate PROMable code.	-P
Generate debug information.	-r

See the *H8 Command Line Interface Guide* for more information.

COMPILATION INTERNAL ERROR MESSAGES

A compiler internal error message indicates that there has been a serious and unexpected failure due to a fault in the compiler itself, for example, the failure of an internal consistency check. After issuing a self-explanatory message, the compiler terminates.

Internal errors should normally not occur and should be reported to the IAR Systems technical support group. Your report should include all possible information about the problem and preferably also a disk with the program that generated the internal error.

COMPILATION ERROR MESSAGES

The following table lists the compilation error messages:

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
0	Invalid syntax	The compiler could not decode the statement or declaration.
1	Too deep <code>#include</code> nesting (max is 10)	Fatal. The compiler limit for nesting of <code>#include</code> files was exceeded. One possible cause is an inadvertently recursive <code>#include</code> file.
2	Failed to open <code>#include</code> file 'name'	Fatal. The compiler could not open an <code>#include</code> file. Possible causes are that the file does not exist in the specified directories (possibly due to a faulty <code>-I</code> prefix or <code>C_INCLUDE</code> path) or is disabled for reading.
3	Invalid <code>#include</code> filename	Fatal. The <code>#include</code> filename was invalid. Note that the <code>#include</code> filename must be written <code><file></code> or <code>"file"</code> .
4	Unexpected end of file encountered	Fatal. The end of file was encountered within a declaration, function definition, or during macro expansion. The probable cause is bad <code>()</code> or <code>{ }</code> nesting.
5	Too long source line (max is 512 chars); truncated	The source line length exceeds the compiler limit.
6	Hexadecimal constant without digits	The prefix <code>0x</code> or <code>0X</code> of hexadecimal constant was found without following hexadecimal digits.
7	Character constant larger than "long"	A character constant contained too many characters to fit in the space of a long integer.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
8	Invalid character encountered: '\xhh'; ignored	A character not included in the C character set was found.
9	Invalid floating point constant	A floating-point constant was found to be too large or have invalid syntax. See the ANSI standard for legal forms.
10	Invalid digits in octal constant	The compiler found a non-octal digit in an octal constant. Valid octal digits are: 0-7.
11	Missing delimiter in literal or character constant	No closing delimiter ' or " was found in character or literal constant.
12	String too long (max is 509)	The limit for the length of a single or concatenated string was exceeded.
13	Argument to #define too long (max is 512)	Lines terminated by \ resulted in a #define line that was too long.
14	Too many formal parameters for #define (max is 127)	Fatal. Too many formal parameters were found in a macro definition (#define directive).
15	',' or ')' expected	The compiler found an invalid syntax of a function definition header or macro definition.
16	Identifier expected	An identifier was missing from a declarator, goto statement, or pre-processor line.
17	Space or tab expected	Pre-processor arguments must be separated from the directive with tab or space characters.
18	Macro parameter 'name' redefined	The formal parameter of a symbol in a #define statement was repeated.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
19	Unmatched <code>#else</code> , <code>#endif</code> or <code>#elif</code>	Fatal. A <code>#if</code> , <code>#ifdef</code> , or <code>#ifndef</code> was missing.
20	No such pre-processor command: 'name'	<code>#</code> was followed by an unknown identifier.
21	Unexpected token found in pre-processor line	A pre-processor line was not empty after the argument part was read.
22	Too many nested parameterized macros (max is 50)	Fatal. The pre-processor limit was exceeded.
23	Too many active macro parameters (max is 256)	Fatal. The pre-processor limit was exceeded.
24	Too deep macro nesting (max is 100)	Fatal. The pre-processor limit was exceeded.
25	Macro 'name' called with too many parameters	Fatal. A parameterized <code>#define</code> macro was called with more arguments than declared.
26	Actual macro parameter too long (max is 512)	A single macro argument may not exceed the length of a source line.
27	Macro 'name' called with too few parameters	A parameterized <code>#define</code> macro was called with fewer arguments than declared.
28	Missing <code>#endif</code>	Fatal. The end of file was encountered during skipping of text after a false condition.
29	Type specifier expected	A type description was missing. This could happen in <code>struct</code> , <code>union</code> , prototyped function definitions/declarations, or in K&R function formal parameter declarations.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
30	Identifier unexpected	There was an invalid identifier. This could be an identifier in a type name definition like: <code>sizeof(int*ident);</code> or two consecutive identifiers.
31	Identifier 'name' redeclared	There was a redeclaration of a declarator identifier.
32	Invalid declaration syntax	There was an undecodable declarator.
33	Unbalanced '(' or ')' in declarator	There was a parenthesis error in a declarator.
34	C statement or func-def in #include file, add "i" to the "-r" switch	To get proper C source line stepping for #include code when the C-SPY debugger is used, the <code>-ri</code> option must be specified. Other source code debuggers (that do not use the UBROF output format) may not work with code in #include files.
35	Invalid declaration of "struct", "union" or "enum" type	A struct, union, or enum was followed by an invalid token(s).
36	Tag identifier 'name' redeclared	A struct, union, or enum tag is already defined in the current scope.
37	Function 'name' declared within "struct" or "union"	A function was declared as a member of struct or union.
38	Invalid width of field (max is <i>nn</i>)	The declared width of field exceeds the size of an integer (<i>nn</i> is 16 or 32 depending on the target processor).

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
39	<code>','</code> or <code>;'</code> expected	There was a missing <code>,</code> or <code>;</code> at the end of declarator.
40	Array dimension outside of "unsigned int" bounds	Array dimension negative or larger than can be represented in an unsigned integer.
41	Member 'name' of "struct" or "union" redeclared	A member of struct or union was redeclared.
42	Empty "struct" or "union"	There was a declaration of struct or union containing no members.
43	Object cannot be initialized	There was an attempted initialization of typedef declarator or struct or union member.
44	<code>;'</code> expected	A statement or declaration needs a terminating semicolon.
45	<code>']'</code> expected	There was a bad array declaration or array expression.
46	<code>':'</code> expected	There was a missing colon after default, case label, or in <code>?</code> -operator.
47	<code>'('</code> expected	The probable cause is a malformed <code>for</code> , <code>if</code> , or <code>while</code> statement.
48	<code>'),'</code> expected	The probable cause is a malformed <code>for</code> , <code>if</code> , or <code>while</code> statement or expression.
49	<code>','</code> expected	There was an invalid declaration.
50	<code>'{'</code> expected	There was an invalid declaration or initializer.
51	<code>'),'</code> expected	There was an invalid declaration or initializer.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
52	Too many local variables and formal parameters (max is 1024)	Fatal. The compiler limit was exceeded.
53	Declarator too complex (max is 128 '(' and/or '*')	The declarator contained too many (,), or *
54	Invalid storage class	An invalid storage-class for the object was specified.
55	Too deep block nesting (max is 50)	Fatal. The {} nesting in a function definition was too deep.
56	Array of functions	An attempt was made to declare an array of functions. The valid form is array of pointers to functions: <pre>int array [5] (); /* Invalid */ int (*array [5]) (); /* Valid */</pre>
57	Missing array dimension specifier	There was a multi-dimensional array declarator with a missing specified dimension. Only the first dimension can be excluded (in declarations of extern arrays and function formal parameters).
58	Identifier 'name' redefined	There was a redefinition of a declarator identifier.
59	Function returning array	Functions cannot return arrays.
60	Function definition expected	A K&R function header was found without a following function definition, for example: <pre>int f(i); /* Invalid */</pre>
61	Missing identifier in declaration	A declarator lacked an identifier.
62	Simple variable or array of a "void" type	Only pointers, functions, and formal parameters can be of void type.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
63	Function returning function	A function cannot return a function, as in: <code>int f()(); /* Invalid */</code>
64	Unknown size of variable object 'name'	The defined object has unknown size. This could be an external array with no dimension given or an object of an only partially (forward) declared struct or union.
65	Too many errors encountered (>100)	Fatal. The compiler aborts after a certain number of diagnostic messages.
66	Function 'name' redefined	Multiple definitions of a function were encountered.
67	Tag 'name' undefined	There was a definition of variable of enum type with type undefined or a reference to undefined struct or union type in a function prototype or as a sizeof argument.
68	"case" outside "switch"	There was a case without any active switch statement.
69	"interrupt" function may not be referred or called	An interrupt function call was included in the program. Interrupt functions can be called by the run-time system only.
70	Duplicated "case" label: nn	The same constant value was used more than once as a case label.
71	"default" outside "switch"	There was a default without any active switch statement.
72	Multiple "default" within "switch"	More than one default in one switch statement.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
73	Missing "while" in "do" - "while" statement	Probable cause is missing { } around multiple statements.
74	Label 'name' redefined	A label was defined more than once in the same function.
75	"continue" outside iteration statement	There was a continue outside any active while, do ... while, or for statement.
76	"break" outside "switch" or iteration statement	There was a break outside any active switch, while, do ... while, or for statement.
77	Undefined label 'name'	There is a goto label with no label: definition within the function body.
78	Pointer to a field not allowed <pre>struct { int *f:6; /* Invalid */ }</pre>	There is a pointer to a field member of struct or union:
79	Argument of binary operator missing	The first or second argument of a binary operator is missing.
80	Statement expected	One of ? : ,] or } was found where statement was expected.
81	Declaration after statement This could be due to an unwanted ; for example: <pre>int i;; char c; /* Invalid */</pre> Since the second ; is a statement it causes a declaration after a statement.	A declaration was found after a statement.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
82	"else" without preceding "if"	The probable cause is bad {} nesting.
83	"enum" constant(s) outside "int" or "unsigned" "int" range	An enumeration constant was created too small or too large.
84	Function name not allowed in this context	An attempt was made to use a function name as an indirect address.
85	Empty "struct", "union" or "enum"	There is a definition of struct or union that contains no members or a definition of enum that contains no enumeration constants.
86	Invalid formal parameter	There is a fault with the formal parameter in a function declaration.
	Possible causes are:	
	<pre>int f(); /* valid K&R declaration */ int f(i); /* invalid K&R declaration */ int f(int i); /* valid ANSI declaration */ int f(i); /* invalid ANSI declaration */</pre>	
87	Redeclared formal parameter: 'name'	A formal parameter in a K&R function definition was declared more than once.
88	Contradictory function declaration	void appears in a function parameter type list together with other type of specifiers.
89	"..." without previous parameter(s)	... cannot be the only parameter description specified.
	For example:	
	<pre>int f(...); /* Invalid */ int f(int, ...); /* Valid */</pre>	

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
90	Formal parameter identifier missing For example: <pre>int f(int *p, char, float ff) /* Invalid - second parameter has no name */ { /* function body */ }</pre>	An identifier of a parameter was missing in the header of a prototyped function definition.
91	Redeclared number of formal parameters For example: <pre>int f(int,char); /* first declaration -valid */ int f(int); /* fewer parameters -invalid */ int f(int,char,float); /* more parameters -invalid */</pre>	A prototyped function was declared with a different number of parameters than the first declaration.
92	Prototype appeared after reference	A prototyped declaration of a function appeared after it was defined or referenced as a K&R function.
93	Initializer to field of width nn (bits) out of range	A bit-field was initialized with a constant too large to fit in the field space.
94	Fields of width 0 must not be named	Zero length fields are only used to align fields to the next int boundary and cannot be accessed via an identifier.
95	Second operand for division or modulo is zero	An attempt was made to divide by zero.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
96	Unknown size of object pointed to	An incomplete pointer type is used within an expression where size must be known.
97	Undefined "static" function 'name'	A function was declared with static storage class but never defined.
98	Primary expression expected	An expression was missing.
99	Extended keyword not allowed in this context	An extended processor-specific keyword occurred in an illegal context; eg interrupt int i.
100	Undeclared identifier: 'name'	There was a reference to an identifier that had not been declared.
101	First argument of '.' operator must be of "struct" or "union" type	The dot operator . was applied to an argument that was not struct or union.
102	First argument of '->' was not pointer to "struct" or "union"	The arrow operator-> was applied to an argument that was not a pointer to a struct or union.
103	Invalid argument of "sizeof" operator	The sizeof operator was applied to a bit-field, function, or extern array of unknown size.
104	Initializer "string" exceeds array dimension	An array of char with explicit dimension was initialized with a string exceeding array size.
	For example:	
	<pre>char array [4] = "abcde"; /* invalid */</pre>	
105	Language feature not implemented	A constant argument or constant pointer is required for the in-line functions.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
106	Too many function parameters (max is 127)	Fatal. There were too many parameters in function declaration/definition.
107	Function parameter 'name' already declared	A formal parameter in a function definition header was declared more than once.
	For example:	
	<pre>/* K&R function */ int myfunc(i, i) /* invalid */ int i; { } /* Prototyped function */ int myfunc(int i, int i) /* invalid */ { }</pre>	
108	Function parameter 'name' declared but not found in header	In a K&R function definition, the parameter was declared but not specified in the function header.
	For example:	
	<pre>int myfunc(i) int i, j /* invalid - j is not specified in the function header */ { }</pre>	
109	';' unexpected	An unexpected delimiter was encountered.
110	')' unexpected	An unexpected delimiter was encountered.
111	'{' unexpected	An unexpected delimiter was encountered.
112	',' unexpected	An unexpected delimiter was encountered.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
113	':' unexpected	An unexpected delimiter was encountered.
114	'[' unexpected	An unexpected delimiter was encountered.
115	'(' unexpected	An unexpected delimiter was encountered.
116	Integral expression required	The evaluated expression yielded a result of the wrong type.
117	Floating point expression required	The evaluated expression yielded a result of the wrong type.
118	Scalar expression required	The evaluated expression yielded a result of the wrong type.
119	Pointer expression required	The evaluated expression yielded a result of the wrong type.
120	Arithmetic expression required	The evaluated expression yielded a result of the wrong type.
121	Lvalue required	The expression result was not a memory address.
122	Modifiable lvalue required	The expression result was not a variable object or a const.
123	Prototyped function argument number mismatch	A prototyped function was called with a number of arguments different from the number declared.
124	Unknown "struct" or "union" member: 'name'	An attempt was made to reference a non-existent member of a struct or union.
125	Attempt to take address of field	The & operator may not be used on bit-fields.
126	Attempt to take address of "register" variable	The & operator may not be used on objects with register storage class.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
127	Incompatible pointers	There must be full compatibility of objects that pointers point to. In particular, if pointers point (directly or indirectly) to prototyped functions, the code performs a compatibility test on return values and also on the number of parameters and their types. This means that incompatibility can be hidden quite deeply, for example: <pre>char>(*p1)[8](int); char>(*p2)[8](float); /* p1 and p2 are incompatible - the function parameters have incompatible types */</pre> The compatibility test also includes checking of array dimensions if they appear in the description of the objects pointed to, for example: <pre>int(*p1)[8]; int(*p2)[9]; /* p1 and p2 are incompatible - array dimensions differ */</pre>
128	Function argument incompatible with its declaration	A function argument is incompatible with the argument in the declaration.
129	Incompatible operands of binary operator	The type of one or more operands to a binary operator was incompatible with the operator.
130	Incompatible operands of '=' operator	The type of one or more operands to = was incompatible with =.
131	Incompatible "return" expression	The result of the expression is incompatible with the return value declaration.
132	Incompatible initializer	The result of the initializer expression is incompatible with the object to be initialized.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
133	Constant value required	The expression in a case label, <code>#if</code> , <code>#elif</code> , bit-field declarator, array declarator, or static initializer was not constant.
134	Unmatching "struct" or "union" arguments to '?' operator	The second and third argument of the ? operator are different.
135	" pointer + pointer" operation	Pointers may not be added.
136	Redeclaration error	The current declaration is inconsistent with earlier declarations of the same object.
137	Reference to member of undefined "struct" or "union"	The only allowed reference to undefined struct or union declarators is a pointer.
138	"- pointer" expression	The - operator may be used on pointers only if both operators are pointers, that is, <code>pointer - pointer</code> . This error means that an expression of the form <code>non-pointer - pointer</code> was found.
139	Too many "extern" symbols declared (max is 32767)	Fatal. The compiler limit was exceeded.
140	"void" pointer not allowed in this context	A pointer expression such as an indexing expression involved a void pointer (element size unknown).
141	<code>#error 'any message'</code>	Fatal. The pre-processor directive <code>#error</code> was found, notifying that something must be defined on the command line in order to compile this module.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
142	"interrupt" function can only be "void" and have no arguments	An interrupt function declaration had a non-void result and/or arguments, neither of which are allowed.
143	Too large, negative or overlapping "interrupt" [value] in name	Check the [vector] values of the declared interrupt functions.
144	Bad context for storage modifier (storage-class or function)	The no_init keyword can only be used to declare variables with static storage-class. That is, no_init cannot be used in typedef statements or applied to auto variables of functions. An active #pragma memory=no_init can cause such errors when function declarations are found.
145	Bad context for function call modifier	The keywords interrupt, banked, non_banked, or monitor can be applied only to function declarations.
146	Unknown #pragma identifier	An unknown pragma identifier was found. This error will terminate object code generation only if the -g option is in use.
147	Extension keyword "name" is already defined by user	Upon executing: #pragma language=extended the compiler found that the named identifier has the same name as an extension keyword. This error is only issued when compiler is executing in ANSI mode.
148	'=' expected	An sfr-declared identifier must be followed by =value.

<i>No</i>	<i>Error message</i>	<i>Suggestion</i>
149	Attempt to take address of "sfr" or "bit" variable	The & operator may not be applied to variables declared as bit or as sfr.
150	Illegal range for "sfr" or "bit" address	The address expression is not a valid bit or sfr address.
151	Too many functions defined in a single module.	There may not be more than 256 functions in use in a module. Note that there are no limits to the number of declared functions.
152	'.' expected	The . was missing from a bit declaration.
153	Illegal context for extended specifier	

H8-SPECIFIC ERROR MESSAGES

None.

COMPILATION WARNING MESSAGES

The following table lists the compilation warning messages:

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
0	Macro 'name' redefined	A symbol defined with #define was redeclared with a different argument or formal list.
1	Macro formal parameter 'name' is never referenced	A #define formal parameter never appeared in the argument string.
2	Macro 'name' is already #undef	#undef was applied to a symbol that was not a macro.
3	Macro 'name' called with empty parameter(s)	A parameterized macro defined in a #define statement was called with a zero-length argument.

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
4	Macro 'name' is called recursively; not expanded	A recursive macro call makes the pre-processor stop further expansion of that macro.
5	Undefined symbol 'name' in #if or #elif; assumed zero	It is considered as bad programming practice to assume that non-macro symbols should be treated as zeros in #if and #elif expressions. Use either: #ifdef symbol or #if defined (symbol)
6	Unknown escape sequence ('\c'); assumed 'c'	A backslash (\) found in a character constant or string literal was followed by an unknown escape character.
7	Nested comment found without using the '-C' option	The character sequence /* was found within a comment, and ignored.
8	Invalid type-specifier for field; assumed "int"	In this implementation, bitfields may be specified only as int or unsigned int.
9	Undeclared function parameter 'name'; assumed "int"	An undeclared identifier in the header of a K&R function definition is by default given the type int.
10	Dimension of array ignored; array assumed pointer	An array with an explicit dimension was specified as a formal parameter, and the compiler treated it as a pointer to object.
11	Storage class "static" ignored; 'name' declared "extern"	An object or function was first declared as extern (explicitly or by default) and later declared as static. The static declaration is ignored.

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
12	Incompletely bracketed initializer	To avoid ambiguity, initializers should either use only one level of {} brackets or be completely surrounded by {} brackets.
13	Unreferenced label 'name'	Label was defined but never referenced.
14	Type specifier missing; assumed "int"	No type specifier given in declaration – assumed to be int.
15	Wrong usage of string operator ('#' or '##'); ignored	This implementation restricts usage of # and ## operators to the token-field of parameterized macros.
In addition the # operator must precede a formal parameter:		
<pre>#define mac(p1) #p1 /* Becomes "p1" */ #define mac(p1,p2) p1+p2##add_this /* Merged p2 */</pre>		
16	Non-void function: "return" with <expression>; expected	A non-void function definition should exit with a defined return value in all places.
17	Invalid storage class for function; assumed to be "extern"	Invalid storage class for function – ignored. Valid classes are extern, static, or typedef.
18	Redeclared parameter's storage class	Storage class of a function formal parameter was changed from register to auto or vice versa in a subsequent declaration/definition.
19	Storage class "extern" ignored; 'name' was first declared as "static"	An identifier declared as static was later explicitly or implicitly declared as extern. The extern declaration is ignored.

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
20	Unreachable statement(s)	One or more statements were preceded by an unconditional jump or return such that the statement or statements would never be executed.
	For example:	
	break;	
	i = 2;	<i>/* Never executed */</i>
21	Unreachable statement(s) at unreferenced label 'name'	One or more labeled statements were preceded by an unconditional jump or return but the label was never referenced, so the statement or statements would never be executed.
	For example:	
	break;	
	here:	
	i = 2;	<i>/* Never executed */</i>
22	Non-void function: explicit "return" <expression>; expected	A non-void function generated an implicit return. This could be the result of an unexpected exit from a loop or switch. Note that a switch without default is always considered by the compiler to be 'exitable' regardless of any case constructs.
23	Undeclared function 'name'; assumed "extern" "int"	A reference to an undeclared function causes a default declaration to be used. The function is assumed to be of K&R type, have extern storage class, and return int.

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
24	Static memory option converts local "auto" or "register" to "static"	A command line option for static memory allocation caused auto and register declarations to be treated as static.
25	Inconsistent use of K&R function - varying number of parameters	A K&R function was called with a varying number of parameters.
26	Inconsistent use of K&R function - changing type of parameter For example: <code>myfunc (34);</code> <code>myfunc(34.6);</code>	A K&R function was called with changing types of parameters. <code>/* int argument */</code> <code>/* float argument */</code>
27	Size of "extern" object 'name' is unknown	extern arrays should be declared with size.
28	Constant [index] outside array bounds	There was a constant index outside the declared array bounds.
29	Hexadecimal escape sequence larger than "char"	The escape sequence is truncated to fit into char.

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
30	Attribute ignored	Since <code>const</code> or <code>volatile</code> are attributes of objects they are ignored when given with a structure, union, or enumeration tag definition that has no objects declared at the same time. Also, functions are considered as being unable to return <code>const</code> or <code>volatile</code> .

For example:

```
const struct s
{
    ...
}; /* no object declared, const ignored - warning */
const int myfunc(void);
/* function returning const int - warning */
const int (*fp)(void); /* pointer to function
returning const int - warning */
int (*const fp)(void);
/* const pointer to function returning int - OK,
no warning */
```

31	Incompatible parameters of K&R functions	Pointers (possibly indirect) to functions or K&R function declarators have incompatible parameter types.
----	--	--

The pointer was used in one of following contexts:

```
pointer - pointer,
expression ? ptr : ptr,
pointer relational_op pointer
pointer equality_op pointer
pointer = pointer
formal parameter vs actual parameter
```

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
32	Incompatible numbers of parameters of K&R functions	Pointers (possibly indirect) to functions or K&R function declarators have a different number of parameters. The pointer is directly used in one of following contexts: pointer - pointer expression ? ptr : ptr pointer relational_op pointerpointer equality_op pointer pointer = pointer formal parameter vs actual parameter
33	Local or formal 'name' was never referenced	A formal parameter or local variable object is unused in the function definition.
34	Non-printable character '\xhh' found in literal or character constant	It is considered as bad programming practice to use non-printable characters in string literals or character constants. Use \0xhhh to get the same result.
35	Old-style (K&R) type of function declarator	An old style K&R function declarator was found. This warning is issued only if the -gA option is in use.
36	Floating point constant out of range	A floating-point value is too large or too small to be represented by the floating-point system of the target.
37	Illegal float operation: division by zero not allowed	During constant arithmetic a zero divide was found.
38	Tag identifier 'name' was never defined	

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
39	Dummy statement. Optimized away!	<p>Redundant code found. This usually indicates a typing mistake in the user code or it might also be generated when using macros which are a little bit too generic (which is not a fault).</p> <p>For example:</p> <pre>a+b;</pre>
40	Possible bug! "If" statement terminated	<p>This usually indicates a typing mistake in the user code.</p> <p>For example:</p> <pre>if (a==b); { <if body> }</pre>
41	Possible bug! Uninitialized variable	<p>A variable is used before initialization (the variable has a random value).</p> <p>For example:</p> <pre>void func (p1) { short a; p1+=a; }</pre>
42		This message is discarded.

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
43	Possible bug! Integer promotion may cause problems. Use cast to avoid it	The rule of integer promotion says that all integer operations must generate a result as if they were of <code>int</code> type if they have a small precision than <code>int</code> and this can sometimes lead to unexpected results.
	For example:	
	<pre>short tst(unsigned char a) { if (-a) return (1); else return (-1); }</pre>	
	This example will always return the value 1 even with the value 0xff. The reason is that the integer promotion casts the variable <code>a</code> to 0x00ff first and then performs a bit not.	
	Integer promotion is ignored by many other C compilers, so this warning may be generated when recompiling an existing program with the IAR Systems compiler.	
44	Possible bug! Single '=' instead of '==' used in "if" statement	This usually indicates a typing mistake in the user code.
	For example:	
	<pre>if (a=1) { <if body> }</pre>	
45	Redundant expression. Example: Multiply with 1, add with 0	This might indicate a typing mistake in the user code, but it can also be a result of stupid code generated by a case tool.

<i>No</i>	<i>Warning message</i>	<i>Suggestion</i>
46	Possible bug! Strange or faulty expression. Example: Division by zero	This usually indicates a bug in the user code.
47	Unreachable code deleted by the global optimizer	Redundant code block in the user code. It might be a result of a bug but is usually only a sign of incomplete code.
48	Unreachable returns. The function will never return	The function will never be able to return to the calling function. This might be a result of a bug, but is usually generated when you have never ending loops in a RTOS system.
49	Unsigned compare always true/false	This indicates a bug in the user code! A common reason is a missing -c compiler switch. For example: <pre>for (uc=10; uc>=0; uc--) { <loop body> }</pre> This is a never ending loop because an unsigned value is always larger than or equal to zero.
51	Signed compare always true/false	This indicates a bug in the user code!

H8-SPECIFIC WARNING MESSAGES

None.

A		C		C compiler options (<i>continued</i>)	
abort (library function)	88	C compiler, features	5	-x	52
abs (library function)	89	C compiler options		-y	44, 213, 215
acos (library function)	89	-A	47, 207	-z	44, 178
alignment (#pragma directive)	175	-a	47	C compiler options summary	31
and_ccr (intrinsic function)	195	-b	56	C library functions. <i>See</i> library functions	
and_exr (intrinsic function)	195	-C	36	C-SPY debugger, using	20
ANSI definition	219	-c	36	calloc (library function)	69, 95
data types	221	-D	45	CCSTR (segment)	213
function declarations	222	-e	11, 36, 181	CDATA (segment)	213
function definition parameters	221	-F	48	ceil (library function)	95
hexadecimal string constants	222	-f	56	CODE (segment)	214
asin (library function)	90	-G	57	code generation options	35
assembler		-g	37, 79	code segment name	59
calling conventions	208	-H	57	codeseq (#pragma directive)	176
interrupt functions	210	-I	53	command line options	55
assembler interface	207	-i	48	compiler version number	191
shell	207	-K	42	configuration	61
assembler source	47	-L	48, 207	const (keyword)	219
assert (library function)	90	-l	49	CONST (segment)	214
assumptions	iv	-m	11, 54, 190	conventions	v
atan (library function)	91	-N	49	cos (library function)	96
atan2 (library function)	91	-n	50	cosh (library function)	96
atof (library function)	92	-O	58	cross-reference list	52
atoi (library function)	92	-o	58	CSTACK (segment)	214
atol (library function)	93	-P	11, 59	CSTARTUP routine	69
		-p	50	CSTR (segment)	215
		-q	11, 50, 207	ctype.h (header file)	80
		-R	59		
		-r	11, 44		
		-S	59	D	
		-s	42, 178	dadd (intrinsic function)	195
		-T	51	data representation	73
		-t	51	data types	73, 221
		-U	52	debug options	44
		-u	43	#define options	45
		-v	55, 190	development cycle	8
		-W	43	diagnostics	225
		-w	43, 187	error messages	227
		-X	51	warning messages	243, 252
B					
bit (extended keyword)	163				
bitfields	75				
bitfields = default (#pragma directive)	175				
bitfields = reversed (#pragma directive)	175				
BITVAR (segment)	213				
bsearch (library function)	94				

INDEX

directives, #pragma 175
disable_max_time (intrinsic function) 196
div (library function) 97
do_byte_eepmov (intrinsic function) 196
do_word_eepmov (intrinsic function) 196
documentation route map 4
dsub (intrinsic function) 197

E

ECSTR (segment) 215
efficient coding 77
Embedded Workbench
 installing 2, 3
 running 2
entry (keyword) 219
enum (keyword) 74, 221
errno.h (header file) 86
error messages 227
exit (library function) 98
exp (library function) 98
exp10 (library function) 99
extended keyword summary 157
extended keywords 163
 bit 163
 far 77, 164
 far_func 76, 165
 huge 77, 166
 interrupt 166
 monitor 168
 near 77, 169
 near_func 76, 170
 no_init 170
 sfr 171
 sfrp 172
 tiny 77, 173
 tiny_func 76, 173
extensions 157

F

fabs (library function) 99
far (extended keyword) 77, 164
far_func (extended keyword) 76, 165
features, C compiler 5
FLIST (segment) 215
float.h (header file) 86
floating point precision, XLINK
 command file 62
 floating-point format 74
 4-byte 74
 8-byte 75
floor (library function) 100
fmod (library function) 100
free (library function) 101
frexp (library function) 101
func_stack_base (intrinsic function) 197
function = default (#pragma directive) 177
function = interrupt (#pragma directive) 177
function = intrinsic (#pragma directive) 178
function = monitor (#pragma directive) 180
function = tiny_func (#pragma directive) 180

G

get_ccr (intrinsic function) 201
get_exr (intrinsic function) 201
get_imask_ccr (intrinsic function) 197
get_imask_exr (intrinsic function) 198
getchar (library function) 66, 102
gets (library function) 102

H

header files 80
 ctype.h 80
 errno.h 86
 float.h 86
 icclbutl.h 81
 limits.h 86
 math.h 81
 setjmp.h 82
 stdarg.h 82
 stddef.h 86
 stdio.h 83
 stdlib.h 83
 string.h 84
heap size 69
hexadecimal string constants 222
huge (extended keyword) 77, 166

I

icclbutl.h (header file) 81
IDATA (segment) 216
IFLIST (segment) 215
include options 53
initialization 69
input and output 66
installation, requirements 1
interrupt (extended keyword) 166
interrupt functions 210
interrupt vectors 210
intrinsic function summary 160
intrinsic function support 191
intrinsic functions
 _args\$ 193
 _argt\$ 194
 and_ccr 195
 and_exr 195
 dadd 195
 disable_max_time 196

intrinsic functions (*continued*)

do_byte_eepmov	196
do_word_eepmov	196
dsub	197
func_stack_base	197
get_ccr	201
get_exr	201
get_imask_ccr	197
get_imask_exr	198
mac	198
macl	198
no_operation	199
or_ccr	199
or_exr	199
ovfaddc	199
ovfaddl	199
ovfaddw	199
ovfnegc	200
ovfnegl	200
ovfnegw	200
ovfshalc	200
ovfshall	200
ovfshalw	200
ovfsubc	201
ovfsubl	201
ovfsubw	201
read_ccr	201
read_exr	201
read_hi_mac	202
read_mac	202
repeat_mac	202
rotlc	202
rotll	202
rotlw	202
rotrc	203
rotrl	203
rotrw	203
set_ccr	205
set_exr	205
set_imask_ccr	203
set_imask_exr	203

intrinsic functions (*continued*)

set_interrupt_mask	204
single_mac	204
sleep	204
tas	205
trapa	205
write_ccr	205
write_exr	205
write_ext_mac	206
write_mac	206
xor_ccr	206
xor_exr	206
intrinsic_on_M02 (global variable)	178
INTVEC (segment)	216
isalnum (library function)	103
isalpha (library function)	104
iscntrl (library function)	104
isdigit (library function)	105
isgraph (library function)	105
islower (library function)	106
isprint (library function)	106
ispunct (library function)	107
isspace (library function)	107
isupper (library function)	108
isxdigit (library function)	109

K

K&R definition	v
Kernighan & Richie definition	219
keywords	
const	219
entry	219
enum	74, 221
signed	220
struct	223
union	223
void	220
volatile	220

L

labs (library function)	109
language extensions	157
language = default (#pragma directive)	181
language = extended (#pragma directive)	181
ldexp (library function)	110
ldiv (library function)	110
library functions	
_formatted_read	152
_formatted_write	153
_medium_read	154
_medium_write	155
_small_write	156
abort	88
abs	89
acos	89
asin	90
assert	90
atan	91
atan2	91
atof	92
atoi	92
atol	93
bsearch	94
calloc	69, 95
ceil	95
cos	96
cosh	96
div	97
exit	98
exp	98
exp10	99
fabs	99
floor	100
fmod	100
free	101
frexp	101
getchar	102

library functions (*continued*)

gets	102
isalnum	103
isalpha	104
iscntrl	104
isdigit	105
isgraph	105
islower	106
isprint	106
ispunct	107
isspace	107
isupper	108
isxdigit	109
labs	109
ldexp	110
ldiv	110
log	111
log10	111
longjmp	112
malloc	69, 112
memchr	113
memcmp	114
memcpy	115
memmove	115
memset	116
modf	117
pow	117
printf	118
putchar	122
puts	123
qsort	124
rand	124
realloc	125
scanf	126
setjmp	129
sin	130
sinh	130
sprintf	131
sqrt	131
srand	132
sscanf	133

library functions (*continued*)

strcat	133
strchr	134
strcmp	135
strcoll	135
strcpy	136
strcspn	137
strerror	137
strlen	138
strncat	138
strncmp	139
strncpy	140
strpbrk	140
strrchr	141
strspn	142
strstr	142
strtod	143
strtok	144
strtol	145
strtoul	146
strxfrm	147
tan	147
tanh	148
tolower	148
toupper	149
va_arg	149
va_end	150
va_list	150
va_start	151
library functions summary	80
limits.h (header file)	86
linker command file	62
list options	46
listings, formatting	48, 50
log (library function)	111
log10 (library function)	111
longjmp (library function)	112

M

mac (intrinsic function)	198
macl (intrinsic function)	198
malloc (library function)	69, 112
math.h (header file)	81
memchr (library function)	113
memcmp (library function)	114
memcpy (library function)	115
memmove (library function)	115
memory models	64
XLINK command file	62
memory = constseg (#pragma directive)	182
memory = dataseg (#pragma directive)	183
memory = default (#pragma directive)	184
memory = far (#pragma directive)	184
memory = huge (#pragma directive)	184
memory = near (#pragma directive)	185
memory = no_init (#pragma directive)	186
memory = shortad (#pragma directive)	186
memset (library function)	116
modf (library function)	117
monitor (extended keyword)	168

N

near (extended keyword)	77, 169
near_func (extended keyword)	76, 170
no_init (extended keyword)	170
NO_INIT (segment)	64, 186, 217
no_operation (intrinsic function)	199
non-volatile RAM	64

O

object filename	58
operators, sizeof	162
optimization	42, 44
or_ccr (intrinsic function)	199
or_exr (intrinsic function)	199
ovfaddc (intrinsic function)	199
ovfaddl (intrinsic function)	199
ovfaddw (intrinsic function)	199
ovfnegc (intrinsic function)	200
ovfnegl (intrinsic function)	200
ovfnewg (intrinsic function)	200
ovfshalc (intrinsic function)	200
ovfshall (intrinsic function)	200
ovfshalw (intrinsic function)	200
ovfsubc (intrinsic function)	201
ovfsubl (intrinsic function)	201
ovfsubw (intrinsic function)	201

P

PATH variable	1
pointers	76
far	77
far_func	76
huge	77
near	77
near_func	76
tiny	77
tiny_func	76
pow (library function)	117
pragma directives. <i>See</i> #pragma directives	
predefined symbols	
__DATE__	189
__FILE__	189
__IAR_SYSTEMS_ICC	189
__LINE__	190
__STDC__	190

predefined symbols (*continued*)

__TID__	190
__TIME__	191
__VER__	191
printf (library function)	67, 118
processor groups	62
PROMable code	59
putchar (library function)	66, 122
puts (library function)	123

Q

qsort (library function)	124
--------------------------	-----

R

rand (library function)	124
RCODE (segment)	217
read_ccr (intrinsic function)	201
read_exr (intrinsic function)	201
read_hi_mac (intrinsic function)	202
read_mac (intrinsic function)	202
realloc (library function)	125
recommendations	77
register I/O	69
repeat_mac (intrinsic function)	202
requirements	1
rotlc (intrinsic function)	202
rotll (intrinsic function)	202
rotlw (intrinsic function)	202
rotrc (intrinsic function)	203
rotrl (intrinsic function)	203
rotrw (intrinsic function)	203
route map	4
run-time library	63
run-time stack	208
running	
a program	20
Embedded Workbench	2

S

scanf (library function)	68, 126
segments	211
BITVAR	213
CCSTR	213
CDATA	213
CODE	214
CONST	214
CSTACK	214
CSTR	215
ECSTR	215
FLIST	215
IDATA	216
IFLIST	215
INTVEC	216
NO_INIT	64, 186, 217
RCODE	217
TEMP	217
UDATA	218
WCSTR	218
set_ccr (intrinsic function)	205
set_exr (intrinsic function)	205
set_imask_ccr (intrinsic function)	203
set_imask_exr (intrinsic function)	203
set_interrupt_mask (intrinsic function)	204
setjmp (library function)	129
setjmp.h (header file)	82
sfr (extended keyword)	171
sfr variables	75
sfrp (extended keyword)	172
shared variable objects	223
shell for interfacing to assembler	207
signed (keyword)	220
silent operation	59
sin (library function)	130
single_mac (intrinsic function)	204
sinh (library function)	130
sizeof (operator)	162
sleep (intrinsic function)	204

INDEX

-
- Special Function Register variables 75
- sprintf (library function) 67, 131
- sqrt (library function) 131
- srand (library function) 132
- scanf (library function) 68, 133
- stack 197
- stack size 65
- stdarg.h (header file) 82
- stddef.h (header file) 86
- stdio.h (header file) 83
- stdlib.h (header file) 83
- strcat (library function) 133
- strchr (library function) 134
- strcmp (library function) 135
- strcoll (library function) 135
- strcpy (library function) 136
- strncpy (library function) 137
- strerror (library function) 137
- string.h (header file) 84
- strlen (library function) 138
- strncat (library function) 138
- strncmp (library function) 139
- strncpy (library function) 140
- strpbrk (library function) 140
- strrchr (library function) 141
- strspn (library function) 142
- strstr (library function) 142
- strtod (library function) 143
- strtok (library function) 144
- strtol (library function) 145
- strtoul (library function) 146
- struct (keyword) 223
- strxfrm (library function) 147
- symbols, undefining 52
-
- T**
- tab spacing 51
- tan (library function) 147
- tanh (library function) 148
- target identifier 190
- target options 54
- tas (intrinsic function) 205
- TEMP (segment) 217
- tiny (extended keyword) 77, 173
- tiny_func (extended keyword) 76, 173
- tolower (library function) 148
- toupper (library function) 149
- trapa (intrinsic function) 205
- tutorial files 7
- tutorials
- adding an interrupt handler 27
 - configuring to suit the target program 9
 - running a program 20
 - using #pragma directives 22
 - using C-SPY 20
- type check 37
-
- U**
- UDATA (segment) 218
- #undef options 52
- union (keyword) 223
-
- V**
- va_arg (library function) 149
- va_end (library function) 150
- va_list (library function) 150
- va_start (library function) 151
- void (keyword) 220
- volatile (keyword) 220
-
- W**
- warning messages 243, 252
- warnings = default (#pragma directive) 187
- warnings = off (#pragma directive) 187
- warnings = on (#pragma directive) 187
- WCSTR (segment) 218
- Workbench
- installing 3
 - running 2
- write_ccr (intrinsic function) 205
- write_exr (intrinsic function) 205
- write_ext_mac (intrinsic function) 206
- write_mac (intrinsic function) 206
-
- X**
- XLINK command file 62
- XLINK options, -A 67
- xor_ccr (intrinsic function) 206
- xor_exr (intrinsic function) 206
-
- SYMBOLS**
- #pragma (directive) 175
- #pragma directive summary 158
- #pragma directives
- alignment 175
 - bitfields = default 175
 - bitfields = reversed 175
 - function = default 177
 - function = interrupt 177
 - function = intrinsic 178
 - function = monitor 180
 - function = tiny_func 180
 - language = default 181
 - language = extended 181
 - memory = constseg 182
 - memory = datasetg 183
 - memory = default 184
 - memory = far 184
-

#pragma directives (<i>continued</i>)				
memory = huge	184	-L (C compiler option)	48, 207	-z (C compiler option) 44
memory = near	185	-l (C compiler option)	49	-z (compiler option) 178
memory = no_init	186	-m (C compiler option)	11, 54, 190	__DATE__ (predefined symbol) 189
memory = shortad	186	-N (C compiler option)	49	__FILE__ (predefined symbol) 189
warnings = default	187	-n (C compiler option)	50	__IAR_SYSTEMS_ICC
warnings = off	187	-O (C compiler option)	58	(predefined symbol) 189
warnings = on	187	-o (C compiler option)	58	__LINE__ (predefined symbol) 190
\$ character	162	-P (C compiler option)	11, 59	__STDC__ (predefined symbol) 190
-A (C compiler option)	47, 207	-p (C compiler option)	50	__TID__ (predefined symbol) 190
-a (C compiler option)	47	-q (C compiler option)	11, 50, 207	__TIME__ (predefined symbol) 191
-A (XLINK option)	67	-R (C compiler option)	59	__VER__ (predefined symbol) 191
-b (C compiler option)	56	-r (C compiler option)	11, 44	_args\$ (intrinsic function) 193
-C (C compiler option)	36	-S (C compiler option)	59	_argt\$ (intrinsic function) 194
-c (C compiler option)	36	-s (C compiler option)	42	_formatted_read (library
-D (C compiler option)	45	-s (compiler option)	178	function) 68, 152
-e (C compiler option)	11, 36, 181	-T (C compiler option)	51	_formatted_write (library
-F (C compiler option)	48	-t (C compiler option)	51	function) 67, 153
-f (C compiler option)	56	-U (C compiler option)	52	_medium_read (library
-G (C compiler option)	57	-u (C compiler option)	43	function) 68, 154
-g (C compiler option)	37, 79	-v (C compiler option)	55, 190	_medium_write (library
-H (C compiler option)	57	-W (C compiler option)	43	function) 67, 155
-I (C compiler option)	53	-w (C compiler option)	43, 187	_small_write (library
-i (C compiler option)	48	-X (C compiler option)	51	function) 67, 156
-K (C compiler option)	42	-x (C compiler option)	52	
		-y (C compiler option)	44, 213, 215	

