IAR C COMPILER FOR THE 78000 GUIDE

DISCLAIMER

The information in this document is subject to change without notice. While the information contained herein is assumed to be accurate, IAR Systems assumes no responsibility for any errors or omissions.

COPYRIGHT NOTICE

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.

© Copyright 1994 IAR Systems AB
© Copyright 1994 NEC Electronics Europe GmbH

TRADEMARKS

C-SPY, ICC, and Micro-Series are trademarks of IAR Systems AB.
MS-DOS is a trademark of Microsoft Corp.
UNIX is a trademark of Bell Laboratories.
MS-DOS/16 Mbyte is a trademark of Rational Systems Inc.

Second edition: July 1994
Part no: ICC78000–2

IAR Systems AB
Islandsgatan 2
P.O. Box 23051
S-750 23 UPPSALA
SWEDEN

ABOUT THIS GUIDE

This guide describes how to install and use the IAR C Compiler for the NEC μPD78000 family of microprocessors.

This guide is divided into two parts: the first part, *IAR 78000 C Compiler*, describes those aspects of the C compiler that are specific to the 78000. The second part, *IAR C Compiler – General Features*, describes features common to all IAR C Compilers.

IAR 78000 C COMPILER

This part consists of the following chapters:

The *Introduction* describes the main features of the IAR C Compiler, and shows how it fits in with the other IAR development tools.

Getting started then shows how to install the C compiler and its associated files, and explains the function of these files.

Using the C compiler describes how to run the 78000 C Compiler, and gives information about file formats it uses.

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.

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.

78000 language extensions describes the extended keywords, `#pragma` keywords, and intrinsic functions specific to the 78000 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.

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

78000 specific command line options summary gives a summary of the additional command line options in the 78000 C Compiler.

78000 specific command line options describes the additional command line options in the 78000 C Compiler.

IAR C COMPILER – GENERAL FEATURES

This part consists of the following chapters:

General command line options summary gives a summary of the C compiler command line options.

General command line options then provides reference information about each command line option.

General C language extensions describes the C language extensions provided for all target processors.

C library functions summary 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.

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

Finally *Diagnostics* lists the compiler warning and error messages.

ASSUMPTIONS

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

- The NEC 78000 processor.
- The 78000 processor assembler language.
- MS-DOS or UNIX depending on your host system.

It 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 user guide uses the following typographical conventions:

Style	*Used for*
`computer`	Text that you type in, or that appears on the screen.
`parameter`	What you should type as part of a command.
`[option]`	An optional part of a command.
reference	A cross-reference to another part of this user guide, or to another guide.

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

CONTENTS

IAR 78000 C COMPILER

INTRODUCTION

The IAR Micro Series is a range of integrated development tools that support a wide choice of target microprocessors. Amongst these tools are the IAR C Compilers – a family of powerful and fast C compilers.

The IAR C Compiler for the NEC 78000 family of microprocessors offers the standard features of the C language, plus many extensions designed to take advantage of the 78000's specific facilities. The compiler is supplied with the IAR Micro Series Assembler for the 78000, with which it is integrated and shares linker and library manager tools.

KEY FEATURES

The IAR C Compiler for the 78000 offers the following key features:

LANGUAGE FACILITIES

- Conformance to the ANSI specification.
- Standard library of functions applicable to embedded systems, with source license option.
- IEEE-compatible floating-point arithmetic.
- Powerful extensions for 78000-specific features, including efficient I/O.
- Generation of fully ROM-compatible code without language restrictions.
- Linkage of user code with assembly routines.
- Long identifiers – up to 255 characters.
- Maximum compatibility with other IAR C Compilers.

PERFORMANCE

- Very fast compilation.
- Memory-based design, avoiding temporary files or overlays.
- Single executable C compiler program file.
- Extensive type-checking at compile time.
- Extensive module interface type checking at link time.
- LINT-like checking of program source.

CODE GENERATION

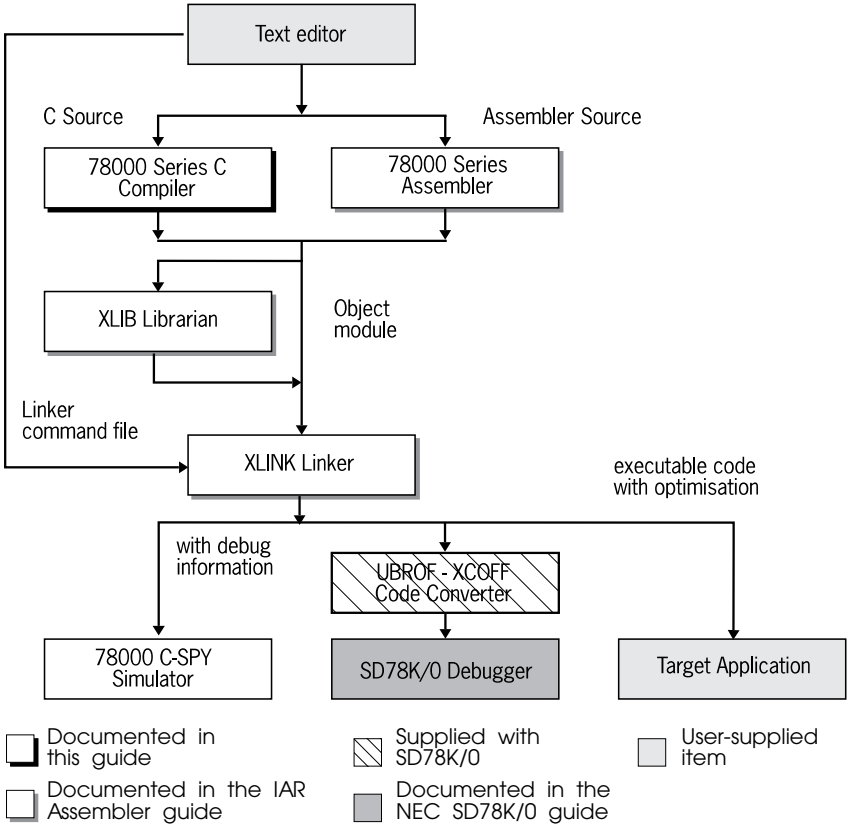
- Selectable optimization levels for code speed and size.
- Comprehensive output options, including relocatable binary, ASM, ASM + C, XREF, etc.
- Easy-to-understand error and warning messages.
- Compatibility with C-SPY high-level debugger, simulator and emulator driver.
- Support for over 20 emulator formats.

TARGET SUPPORT

- Small and banked memory models.
- Flexible variable allocation, including `SFR`, `SFRP`, and `BIT` types.
- Interrupt functions requiring no assembly language.
- A `#pragma` directive to maintain portability while using 78000 extensions.

DEVELOPMENT SYSTEM STRUCTURE

The following diagram shows how the IAR C Compiler is used as part of a complete Micro Series development system:



The text editor may be any standard ASCII editor, such as WordStar, BRIEF, PMATE, or EMACS. The C compiler accepts C source files and produces code module files, normally in the IAR proprietary Universal Binary Relocatable Object Format (UBROF).

These code modules pass to the linker, XLINK, where they may be combined with modules created with the Assembler, and library modules either supplied as standard or created previously by the user using the library manager, XLIB. XLINK and XLIB are supplied and documented as part of the IAR Assembler package.

The output of XLINK is either debuggable code for use in the C-SPY Debugger or an alternative one, or final executable code for use in the target application. This executable code is in any one of many standard formats for use in emulators, EPROM or ROM.

GETTING STARTED

INSTALLATION

This chapter shows you how to install all files from the installation disks supplied, describes the installed files themselves, and lists the file extensions used by the system.

Your should have at least 3 MB of disk space available to install the ICC78000 C Compiler package.

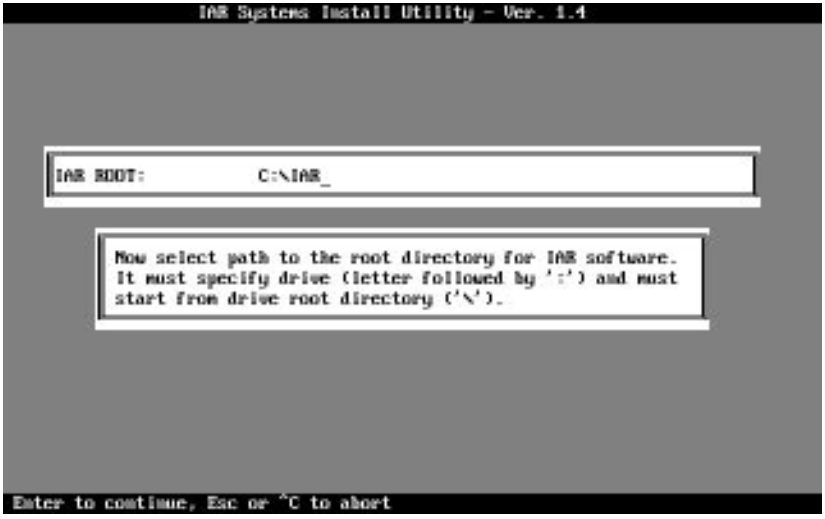
INSTALLATION UNDER MS-DOS

- Ensure your system has MS-DOS 2.11 or higher.
- Insert the installation disk into the floppy disk drive and type:

  `A:\INSTALL` ↵

The startup screen is displayed:

- Press ↵. You will then be prompted to enter the path for installing the IAR subdirectories and files:



By default the files are installed in `C:\IAR`.

- Edit the path, or press ↵ to use the default.

The installation program then decompresses the contents of the installation disks, prompting you for each additional disk.

When decompression is complete, you will see a display of the default paths for each sub-directory into which the files will be installed.

You may edit any of the paths to suit your requirements. You will not normally need to do this, and this guide assumes you have chosen the defaults.

- Press ↵ to proceed.

If you already have some IAR files on the same paths, for example because you are upgrading an existing installation, you will be asked for confirmation before installation proceeds.

The final stage of installation is to manually modify your `autoexec.bat` file. Since the modifications are version-dependent, they are documented in the text file `autoexec.iar` on the directory path you chose (by default, `C:\iar\autoexec.iar`). Open your `autoexec.bat` file and the `autoexec.iar` file in a text editor, follow the instructions in `autoexec.iar` file, and save the modified `autoexec.bat` file.

INSTALLATION UNDER WINDOWS

The IAR C Compiler may be used in an MS-DOS window under Windows. Using an MS-DOS window, follow the instructions given in *Installation under MS-DOS*, page 1-5.

INSTALLATION UNDER UNIX

Follow the separate printed installation documentation supplied with the delivery media.

READ-ME FILES

Your installation includes a number of ASCII-format text files containing recent additional information. Using the default pathnames, they are:

`C:\iar\etc\newclib.doc`	Documentation of additional C library functions.
`C:\iar\icc78000\icc78000.doc`	General information about the C compiler.
`C:\iar\icc78000\global.doc`	General information about the global optimiser.

There are further files associated with the assembler, linker, library manager, and any tools that have been installed separately, such as C-SPY. These are listed in their own guides.

Before proceeding it is recommended that you read all of these files.

INSTALLED FILES

The IAR C Compiler and associated tools use sub-directories and file extensions to make management and operation of them as efficient as possible. This chapter describes these uses and all the IAR files. It refers to the following MS-DOS program files:

Function	*Filename*	*Where it is documented*
78000 C Compiler	`icc78000`	This guide
78000 Assembler	`a78000`	*IAR 78000 Assembler* guide.
IAR Linker	`xlink`	*IAR Linker & Librarian* guide.
IAR Librarian	`xlib`	*IAR Linker & Librarian* guide.
C-SPY Simulator	`cs78000`	*Using C-SPY* guide.

The default installation procedure creates the following directories in `c:\iar`:

Executable files
The `c:\iar\exe` subdirectory holds the MS-DOS executable program files. These correspond to the IAR commands such as the command to run the compiler.

The installation procedure includes an addition to the `autoexec.bat` `PATH` statement, directing MS-DOS to search the `exe` sub-directory for command files. This allows the user to issue an IAR command from any directory.

For details of the contents, see *Exe files*, page 1-11.

Miscellaneous files
The `c:\iar\etc` sub-directory holds miscellaneous files such as read-me files and example sources.

For details of the contents, see *Etc files*, page 1-11.

Source files
The `c:\iar\icc78000` sub-directory holds source files for configuration to the target environment and program requirements, as described in *Configuration*, page 1-43.

For details of the contents, see *ICC78000 files*, page 1-12.

C include files
The `c:\iar\inc` sub-directory holds C include files, such as the header files for the standard C library.

The C compiler searches for include files in the directories given in the `C_INCLUDE` environment variable; see *Files*, page 1-18, and the installation procedure includes the `inc` sub-directory in the definition of this variable in the `autoexec` file; see *Installation under MS-DOS*, page 1-5. This allows the user to refer to an `inc` header file simply by its basename.

For details of the contents, see *INC files*, page 1-13.

Library files
The `c:\iar\lib` sub-directory holds library modules.

The linker searches for library files in the directories given in the `XLINK_DFLTDIR` environment variable (see the *IAR 78000 Assembler* guide), and the installation procedure includes the `lib` sub-directory in the definition of this variable in the `autoexec` file; see *Installation under MS-DOS*, page 1-5. This allows the user to refer to a `LIB` library module simply by its basename.

For details of the contents, see *LIB files*, page 1-15.

Assembler files
The `c:\iar\a78000` sub-directory holds assembler-specific files; see the *IAR 78000 Assembler* guide.

C-SPY files
If you have installed the C-SPY Debugger (supplied separately), there will also be a `c:\iar\cs78000` sub-directory holding the C-SPY-specific files; see the *Using C-SPY* guide.

EXE FILES

This sub-directory contains the following MS-DOS executable program files:

Name	*Function*
`c:\iar\exe\a78000.exe`	78000 Assembler; see the *IAR 78000 Assembler* guide.
`c:\iar\exe\xlib.exe`	IAR Library Manager; see the *IAR 78000 Assembler* guide.
`c:\iar\exe\xlink.exe`	IAR Linker; see the *IAR 78000 Assembler* guide.
`c:\iar\exe\pminfo.exe`	IAR Protected Mode Analyser; see the *IAR 78000 Assembler* guide.
`c:\iar\exe\icc78000.exe`	78000 C Compiler; see *78000 Specific Command line options*, page 1-103.

If you have installed the C-SPY Debugger, this sub-directory will also contain `c:\iar\exe\cs78000.exe`, the 78000 C-SPY Debugger; see the *Using C-SPY* guide.

ETC FILES

This sub-directory contains the following miscellaneous files:

Name	*Function*
`c:\iar\etc\emulator.doc`	Documentation on supported emulators.
`c:\iar\etc\xlink.doc`	Additional information on the linker, XLINK.

Name	*Function*
`c:\iar\etc\newclib.doc`	Information on additional C library functions.
`c:\iar\etc\heap.c`	The source of the heap size control object module. See *Configuration*, page 1-43.
`c:\iar\etc\intwri.c`	The source of the minimal `printf` implementation, as an example. See *C library functions reference*, page 2-45.
`c:\iar\etc\sprintf.c`	The source of the standard `sprintf`, as an example of `va_arg` use. See *C library functions reference*, page 2-45.
`c:\iar\etc\printf.c`	The source of the standard `printf`, as an example of `va_arg` use. See *C library functions reference*, page 2-45.
`c:\iar\etc\frmwri.c`	The source of the **_formatted_write**, used by printf, sprintf. See *C library functions reference*, page 2-45.
`c:\iar\etc\frmrd.c`	The source of the **_formatted_read**. used by scanf and sscanf functions. See *C library functions reference*, page 2-45.
`c:\iar\etc\sieve.c`	The source of the `sieve` example program.

ICC78000 FILES

This sub-directory contains the following configuration starting-point files:

Name	*Function*
`c:\iar\icc78000\icc78000.doc`	Additional information about the C compiler.
`c:\iar\icc78000\lnk780.xcl` `c:\iar\icc78000\lnk780b.xcl`	Linker command files, provided for compatibility with other

Name	*Function*
`c:\iar\icc78000\putchar.c`	The source of `putchar`. See *Configuration*, page 1-43.
`c:\iar\icc78000\getchar.c`	The source of `getchar`. See *Configuration*, page 1-43.
`c:\iar\icc78000\cstartup.s26`	The source of `CSTARTUP`. See *Configuration*, page 1-43.
`c:\iar\icc78000\l07.s26`	The source of the stack check module. See *Configuration*, page 1-43.
`c:\iar\icc78000\global.doc`	Additional information about the global optimiser
`c:\iar\icc78000\iccdemo.bat`	The command file which runs a short demonstaration of the IAR tools for 78000.

INC FILES

This sub-directory contains the following C include files:

Name	*Function*

`c:\iar\inc\string.h`
`c:\iar\inc\float.h`
`c:\iar\inc\math.h`
`c:\iar\inc\stdarg.h`
`c:\iar\inc\limits.h`
`c:\iar\inc\stdio.h`
`c:\iar\inc\stddef.h`
`c:\iar\inc\stdlib.h`
`c:\iar\inc\setjmp.h`
`c:\iar\inc\ctype.h`
`c:\iar\inc\assert.h`
`c:\iar\inc\errno.h`
	Header files for standard C library functions; see *C library functions reference*, page 2-45.

Name	*Function*
`c:\iar\inc\in78000.h`	The source header for intrinsic functions; see *Intrinsic functions*, page 1-63.
`c:\iar\inc\icclbutl.h`	The source header for use by `printf.c`; see *ETC files*, page 1-11.
`c:\iar\inc\iccext.h`	The source header for internal library definitions, not for use by user.
`c:\iar\inc\io7800x.h`	The C source header for I/O addresses of the 7800X processor, see *Language extensions*, page 1-61.
`c:\iar\inc\io7801x.h`	The C source header for I/O addresses of the 7801X processor; see *Language extensions*, page 1-61.
`c:\iar\inc\io78P014.h`	The C source header for I/O addresses of the 78P014 processor; see *Language extensions*, page 1-61.
`c:\iar\inc\io7802x.h`	The C source header for I/O addresses of the 7802X processor, see *Language extensions*, page 1-61.
`c:\iar\inc\io78p024.h`	The C source header for I/O addresses of the 78P024 processor, see *Language extensions*, page 1-61.
`c:\iar\inc\io7804x.h`	The C source header for I/O addresses of the 7804X processor, see *Language extensions*, page 1-61.
`c:\iar\inc\io78p044.h`	The C source header for I/O addresses of the 78P044 processor, see *Language extensions*, page 1-61.

Name	*Function*
`c:\iar\inc\io7805x.h`	The C source header for I/O addresses of the 7805X processor, see *Language extensions*, page 1-61.
`c:\iar\inc\io7806x.h`	The C source header for I/O addresses of the 7806X processor, see *Language extensions*, page 1-61.
`c:\iar\inc\defmsv0.inc`	The assembler source header for small memory model and 7800X processors; see the *IAR 78000 Assembler* guide.
`c:\iar\inc\defmbv0.inc`	The assembler source header for banked memory model and 780XX processors; see the *IAR 78000 Assembler* guide.

LIB FILES

This sub-directory contains all library modules as follows:

Name	*Function*

`c:\iar\lib\cl7800s.r26`
`c:\iar\lib\cl7801s.r26`
`c:\iar\lib\cl7800b.r26`
`c:\iar\lib\cl7801b.r26`
	Library object modules for each combination of memory model (S, B) and processors (7800X, 7800X), see *Configuration*, page 1-43.

FILE TYPES

The IAR C Compiler uses the following default file extensions to identify different types of file:

Extension	*Type of file*	*Output from*	*Input to*
`.doc`	ASCII documentation	–	Text editor
`.exe`	MS-DOS program	–	MS-DOS command
`.c`	C program source	Text editor	ICC78000 command
`.h`	C header source	Text editor	ICC78000 `#include`
`.s26`	Asm program source	Text editor	A78000 command
`.inc`	Asm include source	Text editor	A78000 `#include`
`.xcl`	Linker command files	Text editor	XLINK
`.r26`	Object module	ICC78000 A78000	XLINK, XLIB
`.a26`	Target program	XLINK	EPROM, C-SPY, etc.
`.d26`	Target program with debug information	XLINK	C-SPY, etc.

The default extension may be overridden by simply including an explicit extension when the filename is specified.

USING THE C COMPILER

RUNNING THE C COMPILER

The ICC C Compiler is run by a command of the following form:

```
icc78000 [options] [sourcefile] [options]
```

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

PARAMETERS

`options` A list of options separated by one or more space or tab characters.

`sourcefile` The name of the source file.

If no `options` or `sourcefile` is given, the command displays information about the compiler, including a summary of the options and the target-specific file extensions used.

OPERATION UNDER MS-DOS

The command is entered as described above. Filenames are not case sensitive.

OPERATION UNDER UNIX

Filenames are case sensitive so, for example, `program.c` is *not* equivalent to `PROGRAM.C`. Note that the default extension for C source files is lower case `.c`.

FILES

The compilation process involves the following types of file:

SOURCE FILE

Each invocation of the compiler processes the single source file named on the command line.

Its name is of the form:

```
path leafname.ext
```

For example, the filename `\project\program.c` has the path `\project\`, the leafname `program` and the extension .c. If you give no extension in the name, the compiler assumes `.c`.

INCLUDE FILE

Additional source files may be invoked from the main source file through the `#include` directive. The name of the include file may be given in one of two ways:

Standard search sequence

To use the standard search sequence enclose the filename in angled brackets:

```
<file>
```

For example:

```
#include <incfile.h>
```

The standard search sequence is as follows:

- The include filename with successive prefixes set with the `-I` option if any.
- The include filename with successive prefixes set in the environment variable named `C_INCLUDE` if present. Multiple prefixes may be specified by separating them with semicolon; for example:

  ```
  set C_INCLUDE=\usr\proj\;\headers\
  ```

- The include filename by itself.

Note that the compiler simply adds each prefix from `-I` or `C_INCLUDE` to the front of the `#include` filename without interpretation. Hence it is necessary to include any final backslash in the prefix.

Source file path
To search for the file prefixed by the source file path first, enclose the filename in double quotes:

```
"file"
```

For example:

```
#include "incfile.h"
```

For example, with a source file named `\project\prog.c`, the compiler would first look for the file `\project\incfile.h`. If this file is not found, the compiler continues with the standard search sequence as if angle brackets had been used.

ASSEMBLY SOURCE FILE

The compiler is capable of generating an assembly source file for assembly using the appropriate IAR Assembler. The name is the source file leafname plus the extension `.s26` for assembly sources.

Assembly source file generation is controlled by the `-a` and `-A` options.

OBJECT FILE

The compiler sends the generated code to the object file whose name is, by default, the source file leafname plus the extension `.r26` for object modules.

If any errors occurs during compilation, the object file is deleted. Warnings do not cause the object file to be deleted.

LIST FILE

The compiler can generate a compilation listing, normally to a file with the same leafname as the source, but with the extension `.lst`.

COMMAND FILE

The compiler can accept options and source filename from a command file, as well as from the command line itself. Command files have the extension `.xcl` by default.

TUTORIAL

This chapter provides a tutorial for users new to the IAR C Compiler package. It demonstrates:

- A typical development cycle.
- How to organize the files for a project.
- How to compile and link a simple program.
- How to use the following 78000-specific features: #pragma directives, provided header files, `sfr` variables, `bit` variables and interrupt functions

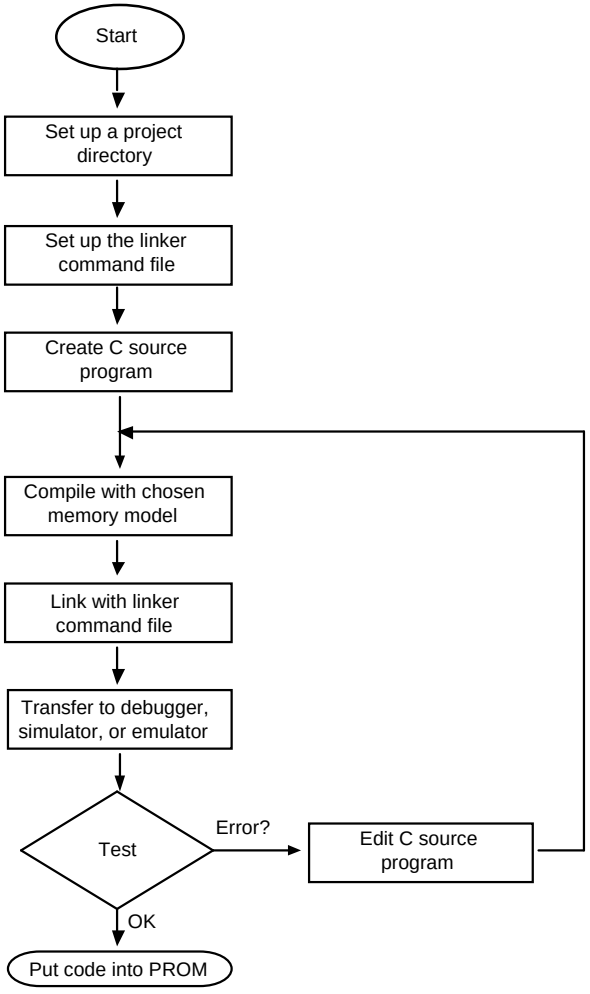
It assumes you are familiar with the C language in general.

You must have already installed the IAR C Compiler for MS-DOS as discussed in the previous chapter.

If you are using a HLL debugger like C-SPY Simulator or SD78K/0, you may follow this tutorial by running the program on your PC resp. emulator, following the instructions given in the documentation supplied with the corresponding debugger.

TYPICAL DEVELOPMENT CYCLE

Development will normally follow the cycle illustrated below:



The following tutorial follows this cycle except for the debug session.

CREATING A PROJECT DIRECTORY

The user files for a particular project are best kept in one directory, separate from other projects and the IAR system files.

Create a project directory by entering the command:

```
mkdir c:\tutorial ↵
```

Select the project directory by entering the command:

```
cd c:\tutorial ↵
```

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

CONFIGURING TO SUIT THE TARGET PROGRAM

Each project needs a linker command file containing details of the target system's memory map. To create this, first copy the linker command file template supplied:

```
copy c:\iar\icc78000\lnk780.xcl ↵
```

This creates a copy called `lnk780.xcl` in your project directory.

Before you edit the linker command file, you need the following items of information about the target system and program requirements:

- The locations of ROM and RAM.

  For this tutorial, use the following locations, which are appropriate to a typical target system:

ROM:	`0x0000` to `0x7FFF`
RAM:	`0xFB00` to `0xFE1F`
Short address RAM:	`0xFE20` to `0xFEDF`

If you are using C-SPY, you could actually specify any reasonable ROM and RAM addresses and C-SPY will automatically simulate them.

If you are using SD78K/0, you have to take into consideration the ROM and RAM areas permitted in the real target device.

- Whether the code will all fit in the ROM available (which allows use of the most efficient, small memory model).

  The tutorial program is small and therefore it will all fit within the ROM specified above.

- The amount of RAM required for the stack.

  The tutorial program has few dynamic variables and no deep nesting of function calls, therefore a 128 (`0x80`) byte stack is sufficient.

Now edit your file `c:\tutorial\lnk780.xcl` using a text editor, following the instructions given in the file to enter these items of information.

Note that these decisions are not permanent: they can be altered later on in the 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 1-50. For detailed information on choosing stack size, see *Stack size*, page 1-51.

SELECTING A LIBRARY FILE

Library selection involves two choices:

Memory model	small or banked.
Processor type	7800X or 780XX

See *Memory model*, page 1-44, *Processor type*, page 1-116, for more details on each of these choices.

Our tutorial program contains only a small amount of code, and therefore requires only the small memory model. The processor to compile for is the 78P014. The appropriate library file for this combination is `cl7801s`.

See *LIB files*, page 1-15, for details of the other library filenames.

CREATING A PROGRAM

The first program is a simple program using just standard C facilities. It repeatedly calls a function that increments a variable. The loop program demonstrates how to compile, link, and run a program.

Using a text editor, enter the source of the loop program:

```
int call_count=0;

/***********************************
 *                                 *
 *        Start of code            *
 *                                 *
 ***********************************/

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

void main(void)
{
 while (1)
 {
   do_foreground_process();
 }
}
```

Save the source as the file `prog.c`.

COMPILING THE PROGRAM

To compile the program, enter the command:

```
icc78000 prog -v2 -ms ↵
```

The `-ms` option selects the small memory model, the `-v2` option selects the 78P014.

If you are going to debug the program you have to add the debugger specific command line options.

Debugging using C-SPY:

```
icc78000 prog -v2 -ms -r ↵
```

Debugging using SD78K/0:

```
icc78000 prog -v2 -ms -rr ↵
```

This creates an object module called `prog.r26`.

LINKING THE PROGRAM

To link the program, enter the command:

```
xlink prog -f lnk780 ↵
```

The `-f` option specifies your linker command file `lnk780`.

If you are going to debug the program you have to add the debugger specific command line options.

Debugging using C-SPY:

```
xlink prog -f lnk780 -r ↵
```

The result of linking is a code file called `aout.d26`.

Debugging using SD78K/0:

```
xlink prog -f lnk780 -r -Y# ↵
```

The result of linking is a code file called `aout.d26`. This debug file in UBROF file format has to be converted into XCOFF file format using the UBROF to XCOFF conversion utility:

```
ubr2xcof aout.d26 ↵
```

The result of linking is a code file called `aout.lnk`.

RUNNING THE PROGRAM

To run the program using C-SPY or SD78K/0, please follow the instructions given in the User's Manual for the corresponding debugger.

EXTENDING THE PROGRAM

We shall now extend the loop program to access the A/D converter built in to the 78P014 microprocessor. The resultant program accepts input from A/D conversion result register and stores the data in a buffer. This A/D conversion program demonstrates the use of the `#pragma` directive, inclusion of supplied header files, use of intrinsic functions and use of bit variables.

The following is a complete listing of the A/D conversion program. The lines that have been added to the loop program are marked with a vertical bar, so, using a text editor, just add these marked lines to the source of the loop program in your `prog.c` file.

```
#pragma language=extended        /* enable use of extended
                                 keywords */
#include <io78p014.h>            /* include sfr definitions
                                 for IO registers */

bit AD_Ready = IF0H.3 ;          /* define bit variable */

/* mode register bits */
#define ADIS0        (0x01)      /* conversion channel 0 */
#define EnableAD     (0x80)      /* start A/D conversion */

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

int call_count=0;
```

```
/***********************************
 *                                 *
 *        Start of code            *
 *                                 *
 ***********************************/
/*
   Return 0 if no conversion result available
        <>0 if data now in ADCR register
 */

int ad_ready(void)
{
 return AD_Ready ;
}

/*
  Data reader: poll status register until ready, return
  data.
*/

 unsigned char read_data(void)
 {
  while (!ad_ready());          /* wait for data */
  return ADCR;                  /* return A/D converted
                                data */
 }

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

```
void main(void)
{
 /* Initialize A/D converter */

  ADIS = ADISO ;

   ADM = EnableAD + 1 ;

 /* now loop forever, taking input when ready */
  while (1)
  {
    if (ad_ready())
    {
      buffer[buffindex++]=read_data();

      if (buffindex==buffsize) { /* process full buffer */
         buffindex=0;            /* simple processing:
      }                          discard data! */
     }
    do_foreground_process();
  }
 }
```

The first line in the program is:

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

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 section *#pragma directive summary*, page 1-62.

The second line is:

```
#include <io78P014.h>            /* include sfr definitions
                                 for IO registers */
```

The file `io78P014.h` includes definitions for all I/O registers for the 78P014 microprocessor. Inside this file, all sfr registers are defined using the `sfr` extended keyword, so if you use this or an alternative header file, you will never need to use the `sfr` keyword directly.

The full list of C source header files for processor I/O is given in *Inc files*, page 1-13.

The third line is:

```
bit AD_Ready = IF0H.3 ;   /* define bit variable */
```

This defines a bit variable `AD_Ready` to be bit number 3 of the location `IF0H`. `IF0H` is one of the Interrupt request flag registers defined in the included header file. Bit variables are described in *Extended keyword reference* page 1-65.

COMPILING AND LINKING THE A/D PROGRAM

Compile and link the program as before:

```
icc78000 prog -v2 -ms ↵
xlink prog -f lnk780 -r ↵
```

If you are going to debug the program you have to add the debugger specific command line options.

Debugging using C-SPY:

```
icc78000 prog -v2 -ms -r ↵
xlink prog -f lnk780 -r ↵
```

Debugging using SD78K/0:

```
icc78000 prog -v2 -ms -rr ↵
xlink prog -f lnk780 -r -Y# ↵
ubr2xcof aout.d26 ↵
```

RUNNING THE PROGRAM

To run the program using C-SPY or SD78K/0, please follow the instructions given in the User's Manual for the corresponding debugger.

ADDING AN INTERRUPT HANDLER

We shall now extend the A/D program by adding an interrupt handler. The IAR C Compiler lets you write interrupt handlers directly in C using the `interrupt` keyword. The interrupt we will handle is the 'A/D conversion' software interrupt.

The following is a complete listing of the interrupt program. The lines that have been added to the A/D program are marked with a vertical bar, so as before, just add these marked lines to the source of the serial program in your `prog.c` file.

```
#pragma language=extended        /* enable use of extended
                                 keywords */
#include <io78p014.h>            /* include sfr definitions
                                 for IO registers */
#include <in78000.h>             /* include for intrinsic
                                 functions */
bit AD_Ready = IF0H.3 ;          /* define bit variable */

/* mode register bits */
#define ADIS0        (0x01)      /* conversion channel 0 */
#define EnableAD     (0x80)      /* start A/D conversion */

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

int call_count=0;
/**********************************
 *                                *
 *        Start of code           *
 *                                *
 **********************************/
/*
   Return 0 if no conversion result available
        <>0 if data now in ADCR register
 */
```

```
int ad_ready(void)
{
 return AD_Ready ;
}

/*
  Data reader: poll status register until ready, return
data.
*/

 unsigned char read_data(void)
 {
  while (!ad_ready());    /* wait for data */
  return ADCR;      /* return A/D converted data */
 }

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

/*
   Example 'interrupt' handler. Take control of A/D
conversion' completion.
 */

interrupt [INTBRK_vect] void ad_exception(void)
{
  call_count = 0 ;
}
```

```
void main(void)
{
 /* Initialize A/D converter */

  ADIS = ADISO ;

   ADM = EnableAD + 1 ;

 /* now loop forever, taking input when ready */
  while (1)
  {
    if (ad_ready())
    {
      buffer[buffindex++]=read_data();

      if (buffindex==buffsize) { /* process full buffer */
         buffindex=0;            /* simple processing:
                                    discard data! */

       if( buffer[ buffindex -1 ] == 0xFF )
         _OPC( 0xBF ) ;          /* Generate BRK interrupt
                                    */
     }
    do_foreground_process();
  }
 }
```

The first addition is:

```
#include <in78000.h>      /* include for intrinsic
                             functions */
```

This include file is necessary to enable intrinsic functions. Our example will use the intrinsic function `_OPC()` later in the program.

The second additions is:

```
interrupt [INTBRK_vect] void ad_exception(void)
{
  call_count = 0 ;
}
```

This function is an exception handler which is called whenever an software break instruction is executed. The `interrupt` keyword is described in *Extended keyword reference*, page 1-65.

The final addition is:

```
if( buffer[ buffindex -1 ] == 0xFF )
  _OPC( 0xBF ) ;                /* Generate BRK interrupt
                                */
```

This line inserts a 78000 assembler instruction using the intrinsic function `_OPC()`. The parameter `0xBF` is the op-code for a 78000 `BRK` instruction which generates a software break. The advantage of intrinsic functions is that the compiler itself translates this 'function call' to the corresponding machine instruction without any overhead of `CALL` or `RET` instructions.

The intrinsic functions require the `in78000.h` file to be included.

COMPILING AND LINKING THE PROGRAM

Compile and link the program as before:

```
icc78000 prog -v2 -ms ↵
xlink prog -f lnk780 -r ↵
```

If you are going to debug the program you have to add the debugger specific command line options.

Debugging using C-SPY:

```
icc78000 prog -v2 -ms -r ↵
xlink prog -f lnk780 -r ↵
```

Debugging using SD78K/0:

```
icc78000 prog -v2 -ms -rr ↵
xlink prog -f lnk780 -r -Y# ↵
ubr2xcof aout.d26 ↵
```

RUNNING THE PROGRAM

To run the program using C-SPY or SD78K/0, please follow the instructions given in the User's Manual for the corresponding debugger.

USING ADDITIONAL MEMORY

If the target system has external RAM in addition to the internal RAM in the 78000 device, the C program can make use of this for data storage. For example, our current program has a small buffer which fits into the internal RAM. A large buffer may not fit into the internal RAM, but it can be moved to external RAM as the following program demonstrates.

The program lines that have changed are marked with a bar, so using a text editor, just change these marked lines in the source of the interrupt program in your `prog.c` file.

```
#pragma language=extended        /* enable use of extended
                                 keywords */
#include <io78p014.h>            /* include sfr definitions
                                 for IO registers */
#include <in78000.h>             /* include for intrinsic
                                 functions */
bit AD_Ready = IF0H.3 ;          /* define bit variable */

/* mode register bits */
#define ADIS0        (0x01)      /* conversion channel 0 */
#define EnableAD     (0x80)      /* start A/D conversion */

#define buffsize 0x1000
char buffer[buffsize];
int buffindex=0;

int call_count=0;
```

```
/***********************************
 *                                 *
 *        Start of code            *
 *                                 *
 ***********************************/
/*
   Return 0 if no conversion result available
        <>0 if data now in ADCR register
 */

int ad_ready(void)
{
 return AD_Ready ;
}

/*
  Data reader: poll status register until ready, return
data.
*/

 unsigned char read_data(void)
 {
  while (!ad_ready());    /* wait for data */
  return ADCR;      /* return A/D converted data */
 }

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

/*
   Example 'interrupt' handler. Take control of A/D
conversion' completion.
 */
```

```
interrupt [INTBRK_vect] void ad_exception(void)
{
  call_count = 0 ;
}

void main(void)
{
 /* Initialize A/D converter */

  ADIS = ADIS0 ;

   ADM = EnableAD + 1 ;

 /* now loop forever, taking input when ready */
  while (1)
  {
    if (ad_ready())
    {
      buffer[buffindex++]=read_data();

      if (buffindex==buffsize) { /* process full buffer */
         buffindex=0;            /* simple processing:
                                    discard data! */

       if( buffer[ buffindex -1 ] == 0xFF )
         _OPC( 0xBF ) ;          /* Generate BRK interrupt
                                    */
     }
    do_foreground_process();
  }
 }
```

The change is from:

```
#define buffsize 0xC0
```

to

```
#define buffsize 0x1000
```

This sets a buffer size of `0x1000` bytes – too large to fit in the internal RAM, which is 1024 bytes.

COMPILING THE LARGE BUFFER PROGRAM

Compile the program as before:

```
icc78000 prog -v2 -ms ↵
```

If you are going to debug the program you have to add the debugger specific command line options.

Debugging using C-SPY:

```
icc78000 prog -v2 -ms -r ↵
```

Debugging using SD78K/0:

```
icc78000 prog -v2 -ms -rr ↵
```

LINKING THE LARGE BUFFER PROGRAM

If you try to link the program as before, you will get the following error:

```
Error [16]: Segment DATA1 is too long for segment
definition
```

This occurs because there is insufficient RAM in the linker command file's target description.

Before proceeding, edit the linker command file, following the instructions within it to cover the following additional memory:

RAM: `0xC000` to `0xDFFF`

Save the linker command file and link the program as before by entering:

```
xlink prog -f lnk780 -r ↵
```

If you are going to debug the program you have to add the debugger specific command line options.

Debugging using C-SPY:

```
xlink prog -f lnk780 -r ↵
```

Debugging using SD78K/0:

```
xlink prog -f lnk780 -r -Y# ↵
```

This time the link will complete, confirming that the larger buffer size is available to the program.

RUNNING THE PROGRAM

To run the program using C-SPY or SD78K/0, please follow the instructions given in the User's Manual for the corresponding debugger.

COMPILING AND LINKING THE C-SPY TUTORIAL

A detailed C-SPY tutorial is described in the C-SPY user's manual, page 27 ff. The corresponding debug file is available and already prepared for debugging. The name is `demo.d26` and you can find it in the `\iar\cs78000` subdirectory.

You may modify this tutorial program if you wish. To do so, you have to create a source file `demo.c` which should contain the C source code you like. To compile the program `demo.c`, enter the command:

```
ICC78000 demo.c -v2 -r ↵
```

The `-v2` option selects the μPD78P014 microcontroller as target device and the `-r` option adds debug information to the output file.

To link the program, enter the command:

```
XLINK demo -o demo -f lnk780 -rt ↵
```

The `-o` option selects the debug output file name, the `-f` option selects the xlink command file `lnk780` and the `-rt` option adds debug information and library support functions for C-SPY's TERMINAL I/O functionality to the output file.

The resulting output file `demo.d26` can be loaded into C-SPY:

```
CS78000 -v2 demo.d26 ↵
```

Then you may follow the tutorial described in the C-SPY user's manual page 30 ff.

COMPILING AND LINKING THE SD78K/0 TUTORIAL

Two detailed SD78K/0 tutorials are described in the SD78K/0 primer, page 6-1 ff resp. 8-1 ff. The corresponding source files are available and they must be prepared for debugging. The source file names are `78k0sub.asm` resp. `78k0main.c` and you can find them on the SD78K/0 distribution disk.

To assemble the program `78k0sub.asm`, enter the command:

```
a78000 78k0sub.asm,78k0sub.lst,78k0sub.r26,S ↵
```

The `S` option adds local symbols to the output file.

To link the program, enter the command:

```
xlink 78k0sub -o sample -c78000 -Z(CODE)CODE=1000 -r ↵
```

The `-o` option selects the debug output file name, the `-c` option selects the device family option, the `-Z` option sets the startaddress for the program code segment and the `-r` option adds debug information to the output file.

To convert the program to the debug format for SD78K0, enter the command:

```
ubr2xcof sample.d26 ↵
```

The resulting output file is `sample.lnk`. This module can be loaded into SD78K0. You may follow the tutorial described in the SD78K0 primer, chapter 6.1.2 on page 6-3 ff.

The second tutorial is based on C program, `78k0main.c`.

To compile the program `78k0main.c`, enter the command:

```
icc78000 78k0main.c -v2 -r ↵
```

The `-v2` option selects the μPD78P014 microcontroller as target device an the `-r` option adds debug information to the output file.

To link the program, enter the command:

```
xlink 78k0main -o 78k0main -f lnk780 -r ↵
```

The `-o` option selects the debug output file name, the `-f` option selects the xlink command file `lnk780` and the `-r` option adds debug information to the output file.

To convert the program to the debug format for SD78K0, enter the command:

```
ubr2xcof 78k0main.d26 ↵
```

The resulting output file is `78k0main.lnk`. This module can be loaded into SD78K0. You may follow the tutorial described in the SD78K0 primer, chapter 8.1.2 on page 8-4 ff.

CONFIGURATION

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

INTRODUCTION

Across the range of applications of the 78000 microprocessor there is considerable variation of the hardware environment, such as the amount of ROM and RAM, and of the user program's requirements, eg the amount of stack RAM. This chapter describes how to configure the IAR C Compiler package to support the environment and usage for a given application.

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

Feature	*Configurable element*
Memory model	Compiler option, linker option.
Memory location	Linker command file.
Non-volatile RAM	Linker command file.
Stack size	Linker command file, `cstartup` module.
`putchar` and `getchar` functions	Run-time library module.
`printf/scanf` facilities	Linker command file.
Heap size	Heap library module.
Initialization of hardware and memory	`cstartup` module.

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

RUN-TIME LIBRARY

The library file controls many of the configurable features of the system.

Two major features require alternative run-time libraries for every combination:

Facilities	*Small memory model*	*Banked memory model*
7800X processors	`cl7800s.r26`	`cl7800b.r26`
780XX processors	`cl7801s.r26`	`cl7801b.r26`

By default the library files are in the directory `c:\iar\lib`.

LINKER COMMAND FILE

Two linker command files for the different memory models are available:

Small memory model: `c:\iar\icc78000\lnk780.xcl`

Banked memory model: `c:\iar\icc78000\lnk780b.xcl`

To create a linker command file for a particular project the user first copies the supplied template `c:\iar\icc78000\lnk780.xcl` or `c:\iar\icc78000\lnk780b.xcl`. The user then modifies this file, as described within the file, to specify the details of the target system's memory map.

MEMORY MODEL

The 78000 microprocessor supports a call table mechanism which allows to reduce the code size when function calls are issued. A function call using the call table results in a one byte instruction while the standard function call requires three bytes. The IAR C Compiler makes use of this by supporting two memory models for each calling mechanism. These offer a choice of function

call mechanism, which affects execution speed and code size, versus maximum size of program code:

Standard call mechanism	*Small memory model*	*Banked memory model*
Variable area	< 64 Kbytes	< 64 Kbytes
External RAM	Yes	Yes
Code size	64 Kbytes	> 1Mbyte

The function call mechanism for internal library calls may be selected by the compiler option `-ms` / `-mS` and `-mb` / `-mB`.

CHOOSING A MEMORY MODEL

In the small memory model, default data and default code are both in the 64K address range, meaning that there is less than 64 Kbytes for code, global data, and local data.

In the banked memory model, code accesses is slower due to the use of the bankswitcher, but code may be placed in any available bank, allowing code size to exceed 64 Kbytes. Though any one module's code cannot exceed 64 Kbytes, the use of multiple modules allows code sizes up to 16 Mbytes.

In all models, the stack and global variables must fit into one single 64 Kbyte.

In both standard and banked memory model runtime library functions can be accessed by `CALLT` instruction. The advantage of `CALLT` runtime library calls is the reduction of the program code size of two bytes for every function call, but you have to accept the penalty of some additional instruction cycles for execution of this instruction. Details are described in the 78000 User's Manual.

SPECIFYING THE MEMORY MODEL

The user's program may use only one model at a time, that is, the same model must be used by all user modules and all library modules.

The memory model must be specified to both the compiler and to the linker.

To specify the memory module to the compiler when a user module is compiled, you use one of the following command line options:

Option	*Model*
`-ms`	Small memory model, internal library calls using `CALL` instruction
`-mS`	Small memory model, internal library calls using `CALLT` instruction
`-mb`	Banked memory model, internal library calls using `CALL` instruction
`-mB`	Banked memory model, internal library calls using `CALLT` instruction

For example, to compile `myprog` for the 78P014 with optimization in the small memory model and `CALLT` instruction for internal library calls, use the command:

```
icc78000 myprog -v2 -mS -z9
```

If you include none of the memory model options, the compiler uses the small memory model and `CALL` instruction for the internal library calls.

To specify the memory model to the linker, you select an appropriate library file:

	Small	*Banked*
Command filename	`lnk780.xcl`	`lnk780b.xcl`

For example, to link the module `myprog` (previously compiled for the small memory model) for the banked memory model, you should use the command:

```
xlink myprog -f lnk780b
```

The `-f` option specifies a command filename; see the *IAR 78000 Assembler* guide for details.

CALLT LIBRARY FUNCTION CALLS

When compiler options `-mS` or `-mB` is active, the run-time system reserves the call table vector area for internal use.

Those entries not used by the run-time system is free to be used by the application.

Compiler use of the call table vector area:

Vector	*Function*
40/41	Reserved for CSPY
42/43	?TB_C_LSH_L10
44/45	?TB_UC_DIV_L10
46/47	?TB_SC_DIV_L10
48/49	?TB_SC_CMP_L10
4A/4B	?TB_MULU_L10
4C/4D	?TB_I_LSH_L10
4E/4F	?TB_I_MUL_L10
50/51	?TB_UI_DIV_L10
52/53	?TB_SI_DIV_L10
54/55	?TB_UI_CMP_L10
56/57	?TB_SI_CMP_L10
58/59	?TB_L_MUL_L10
5A/5B	?TB_UL_DIV_L10
5C/5D	?TB_SL_DIV_L10
5E/5F	?TB_L_ADD_L10
60/61	?TB_L_SUB_L10
62/63	?TB_UL_CMP_L10
64/65	?TB_SL_CMP_L10

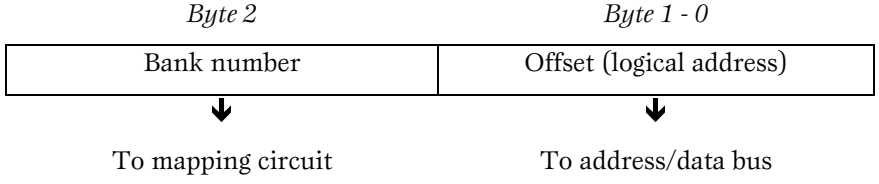
Vector	*Function*
66/67	?TB_UI_LOAD_BITS_L10
68/69	?TB_I_STORE_BITS_L10
6A/6B	?TB_LOAD_A_SP_L10
6C/6D	?TB_STORE_A_SP_L10
6E/6F	?TB_LOAD_AX_SP_L10
70/71	?TB_STORE_AX_SP_L10
72/73	?TB_LOAD_AXBC_SP_L10
74/75	?TB_STORE_AXBC_SP_L10
76/77	?TB_FUNC_ENTER_L10
78/79	?TB_FUNC_DEALL_L10
7A/7B	?TB_FUNC_LEAVE_L10
7C/7D	?TB_WRKSEG_PUSH_L10
7E/7F	?TB_WRKSEG_POP_L10

BANKED MEMORY MODEL

In the `banked` memory model (selected by the `-mb` or `-mB` compiler switch) the code area can be (transparently on the C level) extended with up to 256 blocks of memory while it is identical to the standard memory model in terms of variable allocation and initialisation.

Code block-size can be up to 64K but the requirement to always have an accessible *root* block, for practicular reasons usually limit blocks to 4-32K.

Function addresses are 3 bytes wide and mapped like the following:



Example:

A sample system uses a 16K root PROM and eight 16K PROM banks to create a system with up to 144K of code. An I/O port is used for the purpose of mapping the actual bank to execute. Other ports may be used as shown in the file `l07.s26` which contains the actual switching routines (user-configurable for other mapping hardware schemes).

In the sample system the root PROM was allocated to address 0000-0x7FFF which is most practical since 78000 programs fetch data at location 0x0000 after reset. The root memory contains all vital intrinsic library routines (i.e. support code like floating point arithmetic, rather than user-callable functions) needed by the C program. In addition to that, the root memory also contains C string literals, variable initializers, and optional interrupt handlers.

The "banked" memory blocks of the sample system all start at logical address 0x4000 which means that the `-b` flag to the linker should be set like this:

```
-bCODE=4000,4000,10000
```

The first parameter is the logical address of the initial bank (I/O port = 0), the second parameter shows that 16K banks were used, while the third parameter (increment) gives bank-numbers 0, 1, 2, 3 etc. In the banked area there can only be C callable functions (both library type like `printf` as well as the user-written C routines).

IMPORTANT

- No single module can be larger than the bank-size (over-sized modules result in linker error-messages).
- It is recommended to keep module size considerably smaller than bank-size in order to avoid memory fragmentation (partially filled banks), as the linker only packs complete modules into banks.
- The compiler will in banked mode select the fast "standard" call method if a function is considered as local (i.e. has the storage class static and is not referenced as a function pointer).

- The bank-switching module `l07.s26` must be configured to perform bank-switching. Follow the comments given in this file.
- Interrupt handlers and the correctly configured `l07.r26` bank-switching module have to be located in the root memory, not in any bank.
- Depending on the debugging environment the user may encounter some debugging restrictions when using `banked` functions! For detailed information please refer to the corresponding hardware development tool documentation.

Also see sections concerning the `interrupt`, `banked` and `non_banked` keywords respectively.

MEMORY LOCATION

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

For how to specify the memory address ranges, see the contents of the linker command file template and *XLINK* in the *IAR Assembler, Linker, & Librarian for the 78000*.

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 the user must 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, the user must modify the linker command file. For details how to assign a segment to a given address, see *XLINK* in the *IAR 78000 Assembler* guide.

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.

The only facilities which helps the user to make an estimate of the total required stack size is provided by the NEC IE-78000-R Incircuit Emulator. This Incircuit Emulator allows to specify any range of the 780XX internal RAM area as stack memory and it checks during program execution whether the stack pointer is still inside this range or not. A stack guard error message will occur if the stack pointer exceeds the defined stack area.

The default stack size is set to 128 bytes.

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, the user must provide definitions for these two functions using whatever facilities the hardware environment provides.

The starting-point for the user's routines are supplied, by default, in the files `c:\iar\icc78000\putchar.c` and `c:\iar\icc78000\getchar.c`. 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.
- Compile the modified `putchar` using the appropriate memory model. For example, if the user program uses the small memory model, compile `putchar.c` for the small memory model with the command:

  ```
  icc78000 putchar -v2 -ms -z9
  ```

  This will create an optimized replacement object module file named `putchar.r26`.
- 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 medium-memory-model library, use the command:

  ```
  xlib
  def-cpu 78000
  rep-mod putchar cl7801s
  exit
  ```

  The library module `cl7801s` will now have the modified `putchar` instead of the original.

Note that XLINK allows you to test the modified module before installing it the library by using the `-A` option. See the *IAR 78000 Assembler* guide.

The same procedure is also used for `getchar`.

Note that `putchar` serves as the low-level part of the `printf` function.

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 IAR C standard 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 linker 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_formatter` 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 the user's 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, one alternative smaller version is also provided in the IAR C standard 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 READ FORMATTER VERSION

The selection of a read formatter is made in the linker 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.

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 `c:\iar\etc\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.
- 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`.

The user may wish to modify `cstartup`, for example to initialize special hardware before entry to `main`, or to remove unwanted initialization of variables.

The overall procedure for modifying `cstartup` is as follows:

- Make the required modifications to the assembler source of `cstartup`, supplied by default in the file `c:\iar\icc78000\cstartup.s26`, and save it under the same name.
- Copy one of the provided include files to `devmodel.inc`. The include files select the correct processor and the memory model you want:

  Small memory model: `defmsv0.inc`

  Banked memory model: `defmbv0.inc`

  By default the include files are in the directory `c:\iar\inc`.

- Assemble the modified `cstartup`. This will create a replacement object module file named `cstartup.r26`.
- Add the new `cstartup` module to the appropriate run-time library module, replacing the original.

  For example, to add the new `cstartup` module to the simplest small memory model library, use the command:

  ```
  xlib
  def-cpu 78000
  rep-mod cstartup cl7801s
  exit
  ```

  The library module `cl7801s` will now have the modified `cstartup` instead of the original.

Note that XLINK allows you to test the modified `cstartup` before installing it the library by using the `-C` option. See the *IAR 78000 Assembler* guide for details.

DATA REPRESENTATION

DATA TYPES

The 78000 C Compiler supports all ANSI C basic elements. Variables are stored with the least significant part located at low memory address.

Byte variables are always tightly packed in memory and in structures. Word variables instead are not packed because the NECs 78000 structure requires an even address alignment for any 16-bit data access.

Data type	*Bytes*	*Range*	*Notes*
`bit`	1 bit	0 or 1	see *Extended Keywords*
`sfr`	1	0 to 255	see *Extended Keywords*
`sfrp`	2	0 to 65535	see *Extended Keywords*
`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	-32768 to 32767	
`unsigned short`, `unsigned int`	2	0 to 65535	
`long`	4	-2147483648 to 2147483647	
`unsigned long`	4	0 to 4294967295	
`pointer`	2	0 to 65535	

Data type	*Bytes*	*Range*	*Notes*
`float`	4	±1.18E-38 to ±3.39E+38	
`double`, `long double` (by default)	4	±1.18E-38 to ±3.39E+38 (same as float)	

ENUM TYPE

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

BIT FIELDS

Bit-fields (in unions and structures) can be specified to be based on any of the integral types signed/unsigned char, short or int. This is an extension to ANSI C standard. In expressions, a bit-field will have the same properties (ie signed or unsigned and char, short or int) as the base type. During declaration, bit-field variables are packed in elements of the specified type starting at the LSB position. When a bit-field declarator does not fit within the current element, or if the size of the specified base type differs from the previous bit-field base type, a new element is allocated.

The example below shows the declaration of a number of bit-fields in a structure names 's'.

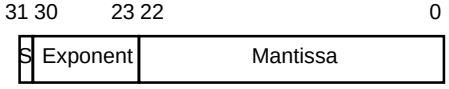
```
struct {

 char a:1; /* Put 'a' in the LSB of a char element */

 char b:5; /* 'b' can be packed together with 'a' */

 char c:4; /* No room for 'c'- allocate a new element */

 int d:3;  /* Diffenetly sized type. Allocate int element
           */

} s ;
```

FLOATING POINT

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

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.

OBJECT POINTERS

Object pointers (pointers to variables) do not point only to one memory type, rather they can point to either short direct address memory (`saddr` or `shortad` pointer) or external data memory area (`near` pointer) of the 78000. The size of a pointer is 2 bytes always.

CODE POINTERS

The code pointers are:

Keyword	*Storage in bytes*	*Restrictions*
`non-banked`	2	May only point into one bank.
`banked`	3	No restrictions.

The `non-banked` pointer is used to reference only functions that are in the default code bank, giving efficient access.

The `banked` pointer can reference any function and is less efficient.

Which of these pointer types is used as the default is determined by the memory model; see *Memory model*, page 1-45.

EFFICIENT CODING

It is important to appreciate the limitations of the 78000 architecture in order to avoid the use of inefficient language constructs. The following is a list of recommendations on how best to use the ICC78000.

- Use 16-bit data types, whenever possible. `long` integers have no direct support in the 78000 architecture. Also note that, according to the ANSI C standard, all data types that are shorter than `int` should undergo integral promotion, ie implicit type conversions, when used in arithmetic expressions.
- Use `unsigned` data types, whenever possible. The 78000 generally performs `unsigned` operations more efficiently than the `signed` counterparts. Especially this applies to type conversions, comparison, array indexing and some arithmetic operations, such as `>>` and `/`.
- Use ANSI prototypes. Function calls to ANSI functions are performed more efficiently than K&R-style functions; see *IAR C Compiler – General Features.*
- Consider using the small memory model. Banked applications will however not be as efficient.
- If the banked memory model is required for your application use the function attribute `non_banked` if possible for local function calls.
- Sensible use of the memory attributes (see *Extended keywords summary*, page 1-61) can enhance both speed and code size in critical applications.
- Avoid using large stack frames. Code efficiency deteriorates in functions with more than 256 bytes of local data. Consider using `static` rather than `auto` storage class for large arrays and structures.
- Use the memory attributes `saddr` or `shortad` if possible. Efficient instructions are available for memory access in short address area.
- To achieve maximum code size optimisation, use the compiler switches `-z9 -W128 -mS`.

LANGUAGE EXTENSIONS

The IAR C Compiler provides a number of powerful extensions that support specific features of the 78000 family of microprocessors.

The 78000 extensions are provided in three ways:

- As extended keywords. By default, the compiler conforms to the ANSI specifications and 78000 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:

ADDRESSING CONTROL

By default the address range in which the compiler places a variable or function 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:

```
near   saddr  shortad
```

or function modifiers:

```
non-banked   banked
```

I/O ACCESS

The program may access the 78000 I/O system using the following data types:

```
sfr    sfrp
```

BIT VARIABLES

The program may take advantage of the 78000 bit-addressing modes by using the following data type:

```
bit
```

NON-VOLATILE RAM

Variables may be placed in non-volatile RAM by using the following data type modifier:

```
no_init
```

INTERRUPT ROUTINES

Interrupt routines may be written in C using the following keywords:

```
interrupt    using  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 bitfields=reversed
#pragma bitfields=default
```

EXTENSION CONTROL

```
#pragma language=extended
#pragma language=default
```

FUNCTION ATTRIBUTE

```
#pragma function=interrupt
#pragma function=monitor
#pragma function=non_banked
#pragma function=banked
#pragma function=default
```

CODESEGMENT USAGE

```
#pragma codeseg(SEG_NAME)
```

MEMORY USAGE

```
#pragma memory=constseg(SEG_NAME)
#pragma memory=dataseg(SEG_NAME)
#pragma memory=near
#pragma memory=saddr
#pragma memory=shortad
#pragma memory=no_init
#pragma memory=default
```

WARNING MESSAGE CONTROL

```
#pragma warnings=on              /* Turn on warnings */
#pragma warnings=off             /* Turn off warnings */
#pragma warnings=default
```

INTRINSIC FUNCTIONS

Intrinsic functions allow very low-level control of the 78000 microprocessor. To use them in a C application, include the header file `in78000.h`.

Most intrinsic functions compile a single 78000 instruction, as follows:

Name	*Instruction*	*Function*
`void _EI(void)`	`EI`	Enable interrupts.
`void _DI(void)`	`DI`	Disable interrupts.
`void _HALT(void)`	`HALT,NOP`	Set HALT mode.
`void _STOP(void)`	`STOP,NOP`	Set STOP mode.
`void _NOP(void)`	`NOP`	No operation
`void _OPC(char constant)`	`--`	Insert one byte constant at the current address .

The `_HALT()` and `_STOP()` intrinsics do not only insert the requested instruction `HALT` resp. `STOP` but a `NOP` instruction as well. This additional `NOP` instruction is necessary due to some device internal structures.

Intrinsics should be used with greatest caution, since they potentially affect the rest of the C-code. For details concerning the effects of the intrinsic functions, see the *78000 User's Manual*.

EXTENDED KEYWORD REFERENCE

This chapter describes the extended keywords in alphabetical order.

The following parameters are used:

Parameter	*What it means*
`storage-class`	Denotes an optional keyword `extern` or `static`.
`declarator`	Denotes a standard C variable or function declarator.

banked

Declares a banked function.

SYNTAX

```
storage-class banked declarator
```

DESCRIPTION

In the small memory model, the default position for functions is within the single data bank. The `banked` keyword indicates that the function is in a different bank, and so must be called by the slower banked method.

CAUTION

Depending on the debugging environment the user may encounter some debugging restrictions when useing `banked` functions. For detailed information pleas refer to the corresponding hardware development tool documentation.

EXAMPLES

The function `my_func` is compiled using the medium memory model. It calls an assembly routine, `my_monitor`, which is located in bank 0. A banked call is therefore required:

```
/* declare my_monitor */
banked void my_monitor(void);

void my_func(void)
{
  /* call the monitor */
  my_monitor() ;
}
```

bit

Declares a bit variable.

SYNTAX – RELOCATABLE ADDRESS

```
storage-class 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 bit-fields.

A `bit` variable can be one of three kinds:

Bit variable type	*Description*
Relocatable address	The variable is one bit of an ordinary relocatable variable.
Fixed address	The variable is one bit of a location at a fixed address between FE20 and FFFF.
sfr	The variable is one bit of an sfr variable.

Bit variables can be used in all places where it is allowed to use other integral types, except:

- As operand to the unary & (address) operator.
- As formal function parameters.
- As struct/union elements.

interrupt

Declare interrupt function or `CALLT` function

SYNTAX

```
storage-class interrupt function-declarator
storage-class interrupt [vector] function-declarator
```

PARAMETERS

`function-declarator`	A void function declarator having no arguments.
`[vector]`	A square-bracketed constant expression yielding the vector address.

DESCRIPTION

The `interrupt` keyword declares a function that is called upon a processor interrupt. The function must be void and have no arguments.

If a vector is specified, the address of an interrupt handler that calls the function is inserted in that vector. The vector address is the offset of the vector from the start of the interrupt vector block, `0x0000`. The vector address must be in the range of `0x0000` to `0x003F`. Predefined vector definitions for popular members of the 78000 family are supplied; see *Installed files*, page 1-8.

Constants for the various interrupt sources are defined in the 78000 specific include files io780*xx*.h files. If no vector is specified, the user must provide an appropriate entry in the vector table (preferably placed in the `cstartup` module) for the interrupt function.

The run-time interrupt handler takes care of saving and restoring processor registers, and returning via the `RETI` instruction, except for vector `3E/3F` that returns with `RETB`. Also register bank switching for interrupt handlers is supported. See `using` keyword later in this chapter.

The compiler disallows calls to 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.

If a vector is specified with a value in the range of `0x0040` to `0x007E`, the address of the function is inserted in the vector table for `CALLT` function calls. Functions declared as `CALLT` accept parameters and may return values as any other function.

When the compiler option `-mS` or `-mB` is in use, there might be a limitation on free resources in the call table area. *See Callt Library Function Calls, page 1-47.*

In banked memory model it is highliy recommended to keep interrupt handlers in a separate file which should be compiled with the option `-RRCODE`, separated form other user functions. This option will give the code segement generated the name RCODE which guarantees that interrupt handlers will not be located in banks but in root memory. *See Banked Memory Model, page 1-48*.

EXAMPLES

```
/* handler for external interrupt 0 */
interrupt [0x24] void ext_0()
 {
 P0 = 6;
 }

/* handler for timer A0 interrupt */
interrupt void timer_A0()
 {
 if (P0.3) start_engine();
 }

/* CALLT function call */
interrupt [0x40] int callt_function( int param )
 {
 if( param == 0 )
  return( TRUE );
 }
```

monitor

Make 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

```
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 modifier.

SYNTAX

```
storage-class near declarator
```

DESCRIPTION

The `near` storage class is the default storage class for any kind of variables. It may be used to override the default storage class after a `#pragma memory=saddr` or `#pragma memory=shortad` directive.

non_banked

Declares a non-banked function.

SYNTAX

```
storage-class non_banked declarator
```

DESCRIPTION

By default, in the banked memory model, all functions are callable from any bank. The `non_banked` keyword indicates that the function is always in the same bank as the caller, and so can be called by the faster unbanked method.

EXAMPLES

Function `foo` is local to one file, and so is only called by functions within the same file. It is therefore always in the same bank as the caller:

```
static non_banked void foo(void)
{
...
}
void foocaller(void)
{
...
foo(); /* call foo by faster non-banked method */
}
```

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 and not to initialize it on start-up.

`no_init` variables are assumed to reside in bank 0. `no_init` variable declarations may not include initializers.

If non-volatile variables are 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 1-50.

EXAMPLES

```
no_init int settings[50];       /* array of non-volatile
                                settings */
no_init int i = 1 ;             /* initializer included -
                                invalid */
```

saddr

Storage modifier.

SYNTAX

```
storage-class saddr declarator
```

DESCRIPTION

By default, the default storage class for any kind of variable (except bit variables) is near. The `saddr` storage modifier may be used to override this.

Variables declared as `saddr` will be located in short address memory from `0xFE20` to `0xFE1F`.

EXAMPLES

```
saddr int var=10;               /* int var in saddr area,
                                   initialised */
saddr char buffer[10] ;         /* array in saddr area */
```

shortad

Storage modifier.

SYNTAX

```
storage-class shortad declarator
```

DESCRIPTION

By default, the default storage class for any kind of variable (except bit variables) is near. The `shortad` storage modifier may be used to override this.

Variables declared as `shortad` will be located in short address memory from `0xFE20` to `0xFE1F`. Variables of type `shortad` cannot be initialised at compile-time and may only be declared on file level.

```
shortad int var;                /* int var in saddr area,
                                   not initialised */
shortad int buffer[10];         /* array in saddr area */
```

sfr

Declare object of one-byte I/O data type.

SYNTAX

```
sfr identifier = constant-expression
```

DESCRIPTION

`sfr` denotes an 78000 SFR-register which:

- Is equivalent to `unsigned char`.
- Can only be directly addressable.
- Resides at a fixed location in the range `0xFF00` to `0xFFFF`.

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.

Predefined `sfr` declarations for popular members of the 78000 family are supplied; see *Installed files*, page 1-8.

EXAMPLES

```
sfr P0 = 0x80;              /* Defines P0 */
void func()
 {
 P0 = 4;                    /* Set entire variable P0 = 00000100
                            */
 P0.2 = 1;                  /* Only affects one bit P0 =
                            XXXXX1XX*/
 if (P0 & 4) printf("ON"); /* Read entire P0 and mask bit 2 */
 if (P0.2) 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 78000 SFR register which:

- Is equivalent to `unsigned int`.
- Can only be directly addressable.
- Resides at a fixed location in the range `0xFF00` to `0xFFFF`.

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.

Predefined `sfrp` declarations for popular members of the 78000 family are supplied; see *Installed files*, page 1-8.

using

Declare interrupt function using another register bank.

SYNTAX

```
storage-class interrupt using [bank] function-declarator
storage-class interrupt [vector] using [bank] function-
declarator
```

PARAMETERS

`function-declarator`	A void function declarator having no arguments.
`[vector]`	A square-bracketed constant expression yielding the vector address.
`[bank]`	A square-bracketed constant expression yielding the alternate bank number.

DESCRIPTION

The `using` keyword declares an interrupt function handler that switches to another register bank before starting the execution of the program code.

The bank parameter must be a constant expression in the range of `0` to `3`. It specifies the 78000 register bank the interrupt handling function will use.

EXAMPLES

```
interrupt [0x24] using [2] void ext_0() /* handler for
external interrupt 0 */
 {
 P0 = 6;
 }
interrupt void using [1] timer_A0() /* handler for timer
A0 interrupt */
 {
 if (P0.3) start_engine();
 }
```

#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

This directive 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

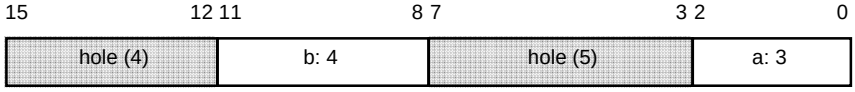
This directive causes the compiler to allocate bitfields starting at the most significant bit of the field, instead of at the least significant bit. The ANSI standard allows the storage order to be implementation-dependent, so you may run into portability problems, which this keyword can be used to avoid.

EXAMPLES

The default layout of

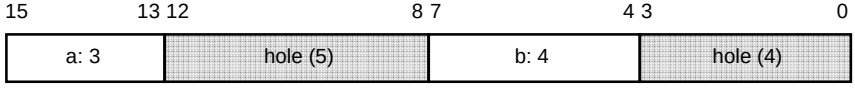
```
struct
{
 short a:3;                    /* a is 3 bits */
 short :5;                     /* this reserves a hole of
                               5 bits */
 short b:4;                    /* b is 4 bits */
} bits;                        /* bits is 16 bits */
```

in memory is:




```
#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 */
```

has the following layout:



codeseg

Directs program code to the named segment by default.

SYNTAX

```
#pragma codeseg (seg_name)
```

DESCRIPTION

This directive directs program code to the named segment by default.

The segment must not be one of the compiler's reserved segment names as listed in *Assembly language interface*, page 1-93.

function = banked

Makes function definitions `banked`.

SYNTAX

```
#pragma function=banked
```

DESCRIPTION

This directive makes subsequent function definitions `banked`. It is an alternative to the function attribute `banked`.

EXAMPLES

```
#pragma function=banked
extern void f1();               /* Identical to extern
                                banked void f1() */
```

CAUTION

Depending on the debugging environment the user may encounter some debugging restrictions when useing `banked` functions. For detailed information pleas refer to the corresponding hardware development tool documentation.

function = default

Restores function definitions to the default type.

SYNTAX

```
#pragma function=default
```

DESCRIPTION

Return function definitions to near or far, as set by the selected memory model. See `function=banked`.

EXAMPLES

```
#pragma function=banked
extern void f1();                /* Identical to extern far
                                 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

```
#pragma function=interrupt
void process_int()              /* an interrupt function */
 {
 }
```

function = monitor

Makes function definitions `monitor`.

SYNTAX

```
#pragma function=monitor
```

DESCRIPTION

This directive makes subsequent function definitions of `monitor` type. It is an alternative to the function attribute `monitor`.

EXAMPLES

```
#pragma function=monitor
void f2()                       /* Will make f2 a monitor
                                   function */
 {
 }
```

function = non_banked

Makes function definitions `non_banked`.

SYNTAX

```
#pragma function=non_banked
```

DESCRIPTION

This directive makes subsequent function definitions `non_banked`. It is an alternative to the function attribute `non_banked`.

EXAMPLES

```
#pragma function=non_banked
extern void f1();               /* Identical to extern non-
                                   banked void f1() */
```

language = default

Restores availability of extended keywords to default.

SYNTAX

```
#pragma language=default
```

DESCRIPTION

This directive returns extended keyword availability to the default set by the `-e` compiler option. See `language=extended`.

language = extended

Makes extended keywords available.

SYNTAX

```
#pragma language=extended
```

DESCRIPTION

This directive makes the extended keywords available regardless of the state of the `-e` compiler option. It is an alternative to the `-e` compiler option. See *Extended keyword reference*, page 1-65, for details.

memory = constseg

Directs constants to the named segment by default.

SYNTAX

```
#pragma memory=constseg (seg_name)
```

DESCRIPTION

This directive 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 as listed in *Assembly language interface*, page 1-93.

memory = dataseg

Directs variables to the named segment by default.

SYNTAX

```
#pragma memory=dataseg (seg_name)
```

DESCRIPTION

This directive directs variables to the named segment by default. The default may be overridden by the memory attributes.

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.

Alternate segments will not be initialised by cstartup.

EXAMPLES

file1.c

```
extern void function(void);
#pragma memory=dataseg(MYSEG)
int variable;                    /* in segment MYSEG */
#pragma memory=default

void main(void)
{
  function() ;
}
```

file2.c

```
#pragma memory=dataseg(MYSEG)
extern int variable;             /* in segment MYSEG */
#pragma memory=default
```

```
void function()
{
  variable = 1 ;
}
```

Leaving out the #pragma in file2.c will cause the linker to give the warning *Type conflict for external/entry variable*. The generated executable file will not work as expected.

memory = default

Restores direction of objects to the default area.

SYNTAX

```
#pragma memory=default
```

DESCRIPTION

This directive restores memory allocation of objects to the default area, as specified by the memory model in use.

memory = near

Direct variables to the default segment by default.

SYNTAX

```
#pragma memory=near
```

DESCRIPTION

This directive directs variables to the default data segment. The default may be overridden by the memory attributes.

The default segment must be linked to coincide with the physical address of the 78000 RAM area; see *Configuration*, page 1-43, for details.

EXAMPLES

```
#pragma memory=no_init
char buffer[1000];              /* in uninitialized memory
                                */
#pragma memory=near
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 = no_init

Direct variables to the `NO_INIT` segment by default.

SYNTAX

```
#pragma memory=no_init
```

DESCRIPTION

This directive 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 *Configuration*, page 1-43, for details.

EXAMPLES

```
#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 = saddr

Direct variables to the `DATA0`, `IDATA0` or `UDATA0` segment by default.

SYNTAX

```
#pragma memory=saddr
```

DESCRIPTION

This directive directs variables to the `DATA0`, `IDATA0` or `UDATA0` segment (depending on the compier command line option `-P`), so that they will reside in short direct addressing RAM. It is an alternative to the memory attribute `near`. The default may be overridden by the memory attributes.

The `DATA0`, `IDATA0` or `UDATA0` segment must be linked to coincide with the physical address of short address RAM; see *Configuration*, page 1-43, for details.

EXAMPLES

```
#pragma memory=saddr
int saddr_var = 1;              /* in short address 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 = shortad

Direct variables to the SHORTAD segment by default.

SYNTAX

```
#pragma memory=shortad
```

DESCRIPTION

This directive directs variables to the SHORTAD segment, so that they will reside in short direct addressing RAM. It is an alternative to the memory attribute near. The default may be overridden by the memory attributes.

No initial values may be supplied in the variable definitions.

The SHORTAD segment must be linked to coincide with the physical address of short address RAM; see *Configuration*, page 1-43, for details.

EXAMPLES

```
#pragma memory=shortad
int shortad_var;                /* in short address 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.

warnings = default

Restores compiler warning output to default state

SYNTAX
```
#pragma warnings=default
```

DESCRIPTION
Return output of compiler warning messages to the default set by the `-w` compiler option. See `#pragma warnings=on` and `#pragma warnings=off`.

---

warnings = off

Turns off output of compiler warnings.

SYNTAX
```
#pragma warnings=off
```

DESCRIPTION
This directive disables output of compiler warning messages regardless of the state of the `-w` compiler option. It is an alternative to the `-w` compiler option.

warnings = on

Turns on output of compiler warnings.

SYNTAX

```
#pragma warnings=on
```

DESCRIPTION

This directive enables output of compiler warning messages regardless of the state of the `-w` compiler option.

ASSEMBLY LANGUAGE INTERFACE

The IAR 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' the user 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:

```
icc78000 shell -A -q -L -z0
```

The `-A` option creates an assembly language output, `-q` includes the C source lines as assembler comments, `-L` creates a listing and `-z0` supresses optimisation.

The result is the listing file `shell.s26` containing the declarations, function call, function return and variable accesses.

The following sections describe the interface in detail.

CALLING CONVENTION

There are two different function parameter passing schemes for the 78000 compiler. These are:

- Prototyped function parameter passing
- Non prototyped function parameter passing

Generally the first parameter is always passed in register(s).

For a Prototyped function up to four parameters can be passed in register(s) depending on parameter sizes in bytes..

Parameters not in register(s) will be pushed on the stack with first parameter not in register(s) at the top of stack.

The *exception* from the above is that as soon as a parameter of type "struct" or "union" is found, that parameter and all following parameters will be put on the stack.

Parameters passed in register(s) will use the following register allocation:

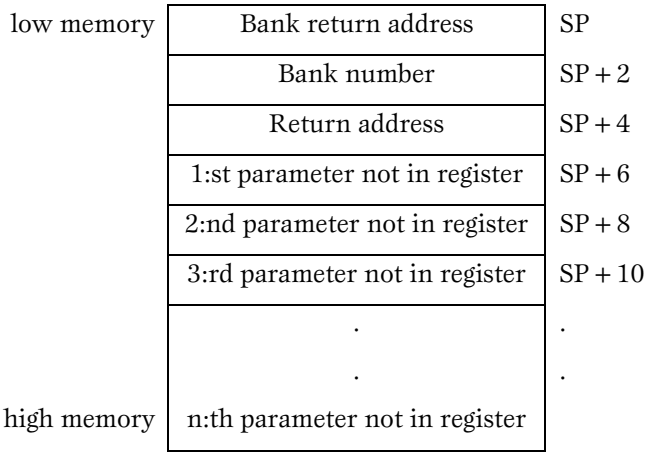
1:st parameter	*2:nd parameter*	*3:rd parameter*	*4:th parameter*
4 bytes (AX,BC)	-	-	-
3 bytes (AX,C)	-	-	-
2 bytes (AX)	2 bytes (BC)	-	-

1:st parameter	*2:nd parameter*	*3:rd parameter*	*4:th parameter*
2 bytes (AX)	1 byte (B)	1 byte (C)	-
1 byte (X)	2 bytes (BC)	-	-
1 byte (X)	1 bytes (A)	2 bytes (BC)	-
1 byte (X)	1 bytes (A)	1 bytes (B)	1 bytes (C)

Immediatly after an entry into a function the stack contains the following:

low memory	Return address	SP
	1:st parameter not in register	SP + 2
	2:nd parameter not in register	SP + 4
	3:rd parameter not in register	SP + 6
	4:th parameter not in register	SP + 8
	.	.
	.	.
high memory	n:th parameter not in register	

The return address in the banked memory model or for a function will also contain the banking return address and the bank number of the invoking function. These are the last objects put on stack immediatly after an entry into a function:



A function is always responsible for deallocating all *own* variables from stack before returning to the caller.

Deallocation of a functions parameters stored on stack is done:

- by **called** function if **prototyped** function
- by **calling** function if **non prototyped** function

FUNCTION RETURN VALUE

Function return values are passed in register(s) except for "structs" and "unions".

The following register allocation is used:

Size	*Register*
1 bit return value	A.0
1 byte return value	A
2 byte return value	AX

Size	*Register*
3 byte return value	AX, B
4 byte return value	AX, BC

A special technique is used for "struct" and "union" return values. The caller reserves an area somewhere in it's own auto space and gives the called function an address to that area as first parameter.

REGISTERS

A function written in assembly language should save all registers used in function at function entry and restore them before return.

SEGMENTS

As can be seen in the linker command files (**lnk*.xcl**) supplied with the compiler, the C system uses a large number of segments. The segments listed in the command files are *reserved* by the run-time system of ICC78000, and may not be used in assembly language programs with the exceptions mentioned in this section.

This information is of no importance for writing C programs but can be useful for programmers mixing C and assembly language. Note that the **lnk*.xcl** files supplied with the compiler package should always be used to guarantee that linking (see section *Linking*) will work as expected

The segments used by the ICC78000 compiler are explained in section *Segment Reference.*

CALLING ASSEMBLY ROUTINES FROM C

An assembler routine that is to be called from C must:

- Conform to the calling convention described above.
- Exit with `RET`.
- Be located in the segment `CODE`.
- Have a `PUBLIC` entry-point label.
- Be prototyped before any call, to allow type checking and promotions of parameters, as in `extern int foo(int i, int j)`.

On entry, SP points to the return address to the calling function.

LOCAL STORAGE ALLOCATION

If the routine needs local storage, it may allocate it in one or more of the following ways:

- On the stack.
- In static workspace, provided of course that the routine is not required to be simultaneously re-usable ("re-entrant").

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 registers. The 78000 automatically saves `PSW` and `PC` on the stack.
- The routine must exit using `RETI`. This automatically restores `PSW` and `PC` from the stack
- The routine must treat all registers and 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, which the supplied linker command files define as the area from address 0x0000 to 0x003F. The interrupt vector 0x001E thus has offset 0x1E within this segment, as used in the following example:

```
        RSEG   INTVEC
        ORG    0x1E              ; Move to vector 0x001E
        WORD   my_int            ; Define interrupt vector
```

The user must place the actual interrupt routine in the RCODE segment, guaranteeing that it will reside in bank 0. The following is an example of a directly installed interrupt routine:

```
        RSEG   RCODE
my_int  PUSH   AX                ; Save the AX register
        MOVW   AX, #0            ; Make sure AX=0
        ...
        POP    AX                ; Restore AX
        RETI                     ; Return from interrupt
```

SEGMENT REFERENCE

The IAR C Compiler places code and data in to named segments which are referred to by the linker. Details of the segments is required for programming assembly language modules, and is 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.

BITVARS

Bit variables.

TYPE
Read-write.

DESCRIPTION
Assembly-accessible.

Holds `bit` variables and can also hold user-written relocatable bit-variables. This segment is NOT initialised by `CSTARTUP` .

CCSTR

String initializers.

TYPE
Read-only.

DESCRIPTION
Assembly-accessible.

Holds C `string` literal initializers when the `-y` (put string literals into variable section) and `-P` (generate PROMable code) compiler option are active.

CDATA0

Variable initializers.

TYPE
Read-only.

DESCRIPTION
Compiler-only.

Holds variable initializers for the variables located in the corresponding `IDATA0` segment. These values are copied over from `CDATA0` to `IDATA0` by `CSTARTUP` during initialization.

CDATA1

Variable initializers.

TYPE
Read-only.

DESCRIPTION
Compiler-only.

Holds variable initializers for the variables located in the corresponding `IDATA1` segment. These values are copied over from `CDATA1` to `IDATA1` by `CSTARTUP` during initialization.

CODE

Code.

TYPE
Read-only.

DESCRIPTION
Assembly-accessible.

Holds user program code, various library routines that can run in alternative banks, and code from assembly language modules.

Note that any assembly language routines included in the `CODE` segment must meet the calling convention of the memory model in use.

CONST

Constants.

TYPE
Read-only.

DESCRIPTION
Assembly-accessible.

Used for storing `const` and `code` objects. Can be used in assembly language routines for declaring constant data.

CSTACK

Stack.

TYPE
Read/write.

DESCRIPTION
Assembly-accessible.

Holds the internal stack.

CSTR

String literals.

TYPE
Read-only.

DESCRIPTION
Assembly-accessible.

Holds C string literals. See the description of the `-y` option (put C string literals into RAM) in *General command line options*, page 2-5.

DATA0

Uninitialized `short address` statics.

TYPE
Read/write.

DESCRIPTION
Compiler-only.

Holds static variables which are not to be zeroed on start-up. These will have been allocated by the compiler, declared `shortad` or created `shortad` by use of the memory `#pragma`, or created manually from assembly language source.

DATA1

Uninitialized statics.

TYPE
Read/write.

DESCRIPTION
Compiler-only.

Holds static variables which are not to be zeroed on start-up.

ECSTR

Writeable string literals.

TYPE
Read/write.

DESCRIPTION
Assembly-accessible.

Holds writeable copies of C string literals when the compiler's `-y` option is active. See the description of the `-y` option (put C string literals into RAM) in *General command line options*, page 2-5.

IDATA0

Initialized `short address` statics.

TYPE
Read/write.

DESCRIPTION
Compiler-only.

Holds static `shortad` variables which have been declared with explicit initial values. Their initial values are copied over from the corresponding segment by `CSTARTUP` during initialization.

IDATA1

Initialized statics.

TYPE
Read/write.

DESCRIPTION
Compiler-only.

Holds static variables which have been declared with explicit initial values. Their initial values are copied over from the corresponding segment by `CSTARTUP` during initialization.

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

Vector handling code.

TYPE
Read-only.

DESCRIPTION
Assembly-accessible.

Used for start-up code and interrupt handlers, that must reside in non-banked memory area.

SHORTAD

Short address memory.

TYPE
Read/write.

DESCRIPTION
Assembly-accessible.

Holds static `saddr` variables which have been declared without explicit initial values. They are set to zero by `CSTARTUP` during initialization. `SHORTAD` can also hold user-written data elements that should initially be set to zero.

TEMP

Autos.

DESCRIPTION
Used for `autos` when compiling with the `-d` option.

UDATA0

Uninitialized `shortad` statics.

TYPE
Read/write.

DESCRIPTION
Assembler-accessible.

Holds static variables which were declared without initial values. ANSI C specifies that such variables be set to zero before they are encountered by the program, so they are set to zero by `CSTARTUP` during initialization. These will have been allocated by the compiler, declared `shortad` or created `shortad` by use of the memory `#pragma`, or created manually from assembly language source.`UDATA0` can also hold user-written data elements that should initially be set to zero.

UDATA1

Uninitialized statics.

TYPE
Read/write.

DESCRIPTION
Assembler-accessible.

Holds static variables which were declared without initial values . ANSI C specifies that such variables be set to zero before they are encountered by the program, so they are set to zero by `CSTARTUP` during initialization. `UDATA1` can also hold user-written data elements that should initially be set to zero.

WCSTR

Writeable string literals.

TYPE
Read/write.

DESCRIPTION
Holds writable copies of the C "string" literals when the `-y` compiler option is active.

WRKSEG

Local register variables.

TYPE
Read/write.

DESCRIPTION
Compiler-only.

Holds the register optimisation memory area when `-W` compiler option is selected.

ZVECT

Initialization.

DESCRIPTION
Used for initialization in `CSTARTUP`.

78000 SPECIFIC COMMAND LINE OPTIONS SUMMARY

The ICC Compiler has an extensive set of target specific command line options that control its operation. Each option consist of a hyphen (-) followed by an option identifier. The position of an option in the command line has no significance in itself.

The options are arranged into the following functional groups:

MEMORY MODEL

`-ms`	Standard. Generates non banked function calls.
`-mb`	Banked. Generates banked external function calls.
`-mS`	Same as `-ms` but using call table calls for internal (run-time) library calls.
`-mB`	Same as `-mb` but using call table calls for internal (run-time) library calls.

PROCESSOR OPTION

`-v0`	Processor option 7800X (default).
`-v1`	Processor option 7801X.
`-v2`	Processor option 78P014.
`-v3`	Processor option 7804X.
`-v4`	Processor option 78P044.
`-v5`	Processor option 7805X.
`-v6`	Processor option 7806X.
`-v7`	Processor option 7802X.
`-v8`	Processor option 78P024.

CODE CONTROL

`-W{rs}`	Set register area size.
`-d`	Force static allocation of auto variables.
`-rr`	Supress variable in register optimization.

78000 SPECIFIC COMMAND LINE OPTIONS

In addition to the general command line options described in *General General command line options*, page 2-5, the 78000 C Compiler has the following options:

-m

Selects memory model.

SYNTAX

`-m[sbSB]`

DESCRIPTION

Use the `-m` option to select the memory model, as follows:

Option	*Memory model*
`s`	Standard (default).
`b`	Banked.
`S`	Standard using call table function calls for internal (run-time) library calls.
`B`	Banked using call table function calls for internal (run-time) library calls.

CAUTION

Depending on the debugging environment the user may encounter some debugging restrictions when useing `banked` functions. For detailed information pleas refer to the corresponding hardware development tool documentation.

For more information see *Memory model*, page 1-44.

Note that all modules of a program must use the same memory model, and must be linked with a library file for that model.

-V

Selects the processor type

SYNTAX

`-v[0-8]`

DESCRIPTION

Use the `-v` option to select the processor type, as follows:

Option	*Processor type*
0	7800X (default)
1	7801X
2	78P014
3	7804X
4	78P044
5	7805X
6	7806X
7	7802X
8	78P024

-W

Set register area size.

SYNTAX

```
-W[rs]
```

DESCRIPTION

The Compiler set register area size option (`-W[rs]`) enables the compiler register variable work area in short direct address memory, segment WRKSEG. The compiler puts auto variables in that area as much as it can. The default maximum register area size, if{rs} not given, is 20 bytes while the maximum {rs} value possible to give is 128 bytes.

Even though only the *actually used number of bytes will be allocated* for each function it is possible that this may increase the code size instead of reduce it, if the function is small, since there is some overhead in managing the short address work area.

-d

Force static allocation of auto variables

SYNTAX

```
-d
```

DESCRIPTION

The Compiler will force variables of storage class auto to storgage class static.

The C compiler will create a segment named TEMP which holds the automatic variables. TEMP can be located anywhere in memory.

When option `-d` is active, the compiled code is no longer reentrant.

-rr

Suppress variable in register optimization

SYNTAX

```
-rr
```

DESCRIPTION

The `-r` option with modifier `r` informs the compiler to store back variables kept in registers to their memory location between statements. The compiler will still keep the value in register(s) if possible. This option/modifier combination is necessary for debugging reason and it is not documented in the compiler sign-on message.

This option must be specified together with the `-Y#` option of the linker when you use the SD78K0 debugger from NEC.

78000 SPECIFIC DIAGNOSTICS

In addition to the error and warning messages described in *Diagnostics*, page 2-155, the 78000 C Compiler has the following error message:

No	*Error message*	*Suggestion*
105	`Language feature not implemented: 'Bit fields SIGNED/UNSIGNED LONG'`	Self explaining sub message
142	`"interrupt" functions can only be "void" and have no arguments`	See section *Interrupt Functions*
143	`Too large, negative or overlapping "interrupt" [value] in 'name'`	Check [vector] values of 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 "typedefs" or applied to "auto" variables of functions. An active "#pragma memoy = no_init" can cause such errors when function declarations are found.
145	`Bad context for function call modifier`	The keywords "interrupt", "non_banked" or "monitor" can only be applied to function declarations.

No	*Error message*	*Suggestion*
146	`Unknown #pragma identifier:'name'`	See section *#pragma Commands*.
147	`Extension keyword "name" is already defined by the user`	This error will occur if a keyword that can serve as an extension keyword is used as an ordinary identifier (when the compiler is executing in ANSI mode) and the directive "#pragma language = extended" is found.
148	`'=' expected`	"sfr"-declared identifiers must be followed by " = value".
149	`Attempt to take address of "sfr", "sfrp" or "bit" variable`	The *unary* &-operator may not be applied to variables declared as "bit", "sfr" or "sfrp".
150	`Illegal range for "sfr", "sfrp" or "bit" address`	The address expression is not a valid "bit", "sfr" or "sfrp" address
152	`'.' expected`	Bad "bit" declaration syntax.
153	`Illegal context for [bit, saddr, shortad, sfr, sfrp, near, no_init] specifier`	This error will be generated due to illegal use of one of the extended keywords.

No	*Error message*	*Suggestion*
154	78000 specific:	Self explaining sub message
	'Not a BIT accessible SFR'	
	'Interrupt function cannot be declared BANKED or MONITOR'	
	'Interrupt function bad USING bank number'	
	'CALLT function cannot be declared USING'	
	'CALLT function cannot be declared BANED or MONITOR'	
	'CALLT function without vector not legal'	
	'CALLT illegal function interrupt vector'	
	'Illegal argument to _DI'	
	'Illegal argument to '_EI'	
	'Illegal argument for _HALT'	
	'Illegal argument for _STOP'	
	'Illegal argument for _NOP'	
	'Argument must exist for _OPC'	
	'Too many arguments for _OPC'	
	'Argument not char constant for _OPC'	

78000 SPECIFIC XLINK ERRORS

In addition to the error and warning messages described in *Diagnostics*, page 2-145, the XLINK has the following error message:

No	*Error message*	*Suggestion*
16	`Function 'name' in module (file) is called from two function trees (with roots 'name1' and 'name2)`	Probable cause: An "interrupt" function calls a function that is also executed by a background program. May lead to execution errors.
17	`Segment 'name' is too large or placed at wrong address`	This warning indiates that a segment holding BIT, SADDR or SHORTAD elements is too large due to too many C variable declarations or is placed at wrong address in the linker command file.
70	`Module module (file) has different memory model than previously linked modules`	Inconsistent use of memory models
71	`Segment 'name' is defined incorrectly (in a bank definition, segment has wrong type, or is mixed with other segment types)`	This error indicates that a faulty linker command file is used.
72	`Segment 'name' must be defined in a-Z definition`	This error should not occur if a properly modified linker command is used.

GENERAL COMMAND LINE OPTIONS SUMMARY

The ICC Compiler has an extensive set of command line options that control its operation. Those options common to all targets are documented in this chapter. In addition there may be options specific to this particular target, in which case these are documented in the chapter *Target specific command line options.*

Each option consist of a hyphen (`-`) followed by an option identifier. Some options are followed by an optional or obligatory argument. If the argument is a file leafname, it must be separated from the option identifier by one or more space or tab characters, for example:

```
-o objfile
```

All other types of arguments (including file prefixes) must immediately follow the identifier, for example:

```
-Opathname
```

The position of an option in the command line has no significance in itself. However in the case of the options `-D` and `-I`, the order of multiple options is important.

The options are arranged into the following functional groups:

FILE CONTROL

`-a file`	Generates assembler source.
`-Aprefix`	Generates assembler source.
`-f file`	Reads command line options from a file.
`-G`	Opens the standard input as source.
`-Iprefix`	Adds an include file search prefix.

`-l file`	Generates a listing.
`-Lprefix`	Generates a listing.
`-o file`	Specifies object filename.
`-Oprefix`	Specifies object filename.

LISTING CONTROL

`-F`	Generates a formfeed after each listed function.
`-i`	Lists included files.
`-pn`	Formats listing into pages.
`-q`	Puts mnemonics in the listing.
`-tn`	Sets the tab spacing.
`-T`	Lists active lines only.
`-x[D][F][T][2]`	Generates a cross-reference list.

CODE CONTROL

`-b`	Makes object a library module.
`-e`	Enables target dependent extensions.
`-Hname`	Sets the object module name.
`-P`	Generates PROMable code.
`-r[012][i][n]`	Generates debug information.
`-Rname`	Sets the code segment name.
`-s[0-9]`	Optimizes for speed.
`-z[0-9]`	Optimizes for size.
`-y`	Initializes strings as variables.

LANGUAGE SPECIFICATION

`-c`	Specifies the interpretation of `char`.
`-C`	Enables nested comments.

`-g[A][O]`	Enables global type check.
`-K`	Enables C++ comments.

MESSAGE CONTROL

`-S`	Sets silent operation of compiler.
`-w`	Disables warnings.
`-X`	Displays C declarations.

USER OPTIONS

`-Dsymb`	Defines a symbol.
`-Dsymb=xx`	Defines a symbol.
`-Usymb`	Undefines a symbol.

GENERAL COMMAND LINE OPTIONS

This chapter lists the C compiler command line options.

-a

Generates assembler source.

SYNTAX

```
-a file
```

DESCRIPTION

Use `-a` to generate assembler source on: `file.sxx`.

By default the compiler does not generate an assembler source. The `-a` 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 appropriate IAR Assembler.

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

The `-a` and `-A` options may not be used together.

-A

Generates assembler source.

SYNTAX

```
-Aprefix
```

DESCRIPTION

Use `-A` to generate assembler source on: `prefix source.sxx`.

By default the compiler does not generate an assembler source . The `-A` option generates an assembler source to a file with the same name as the source leafname but with the target-specific assembly source extension.

The `-A` option may be followed by a `prefix`, which the compiler adds to the filename. This allows the user to redirect the assembly source to a different directory.

The assembler source may be assembled by the appropriate Micro-Series assembler.

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

The `-a` and `-A` options may not be used together.

-b

Makes object a library module.

SYNTAX

```
-b
```

DESCRIPTION

By default the object module is a program object module. Use the `-b` option to make a library object module instead.

-c

Specifies the interpretation of `char`.

SYNTAX

```
-c
```

DESCRIPTION

The ANSI standard specifies that the interpretation of `char` as `unsigned char` or `signed char` is implementation dependent.

By default, the IAR C Compiler treats `char` as equivalent to `unsigned char`. Use `-c` to treat `char` as equivalent to `signed char` for compatibility with other compilers.

Note that the C Library is compiled without `-c`, so if `-c` is used, the type checking enabled by the `-g` or `-r` option may cause unexpected type mismatch warnings from the linker.

-C

Enables nested comments.

SYNTAX

```
-C
```

DESCRIPTION

By default, the compiler issues warnings on finding nested comments. Use `-C` to inhibit these warnings, and allow comments to be nested to any level. This is particularly useful for commenting-out program sections that themselves contain comments.

-D

Defines a symbol.

SYNTAX

```
-Dsymb
-Dsymb=xx
```

DESCRIPTION

The `-Dsymb` option defines a symbol with the value 1 as if the line

```
#define symb 1
```

was included at the start of the source. It provides a mechanism for command line control of the user's own compilation-time options, such as configuration or custom debugging or trace routines. For simple Boolean control variables, it is a more compact mechanism than the more flexible `-Dsymb=xx` option.

The `-Dsymb=xx` option defines a symbol with the specified value as if the line

```
#define symb xx
```

was included at the start of the source.

To include spaces in the expression, surround the whole option by double quotes. For example:

```
"-DEXPR=F + g"
```

is equivalent to:

```
#define EXPR F + g
```

To include a double quote character itself, follow it immediately by a second double quote character. For example:

```
"-DSTRING=""micro proc"""
```

is equivalent to:

```
#define STRING "micro proc"
```

There is no limit on the number of `-D` options used on a single command line.

Command lines can become very long when using the `-D` option, in which case it may be useful to use a command file; see `-f`.

-e

Enables target dependent extensions.

SYNTAX

```
-e
```

DESCRIPTION

Use `-e` to enable extensions that are specific to the particular target. By default these are not enabled.

These extensions are documented in the chapter *Language extensions*.

-f

Reads command line options from a file.

SYNTAX

```
-f file
```

DESCRIPTION

Extends the command line with `file.xcl`.

By default, the compiler looks for command parameters only on the command line itself. To make long command lines more manageable, and to avoid the MS-DOS command line length limit, `-f` may be used to specify a command file, from which the compiler reads command line items as if they had been entered at the position of the `-f` option.

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

If no extension is included in the filename, `.xcl` is assumed.

-F

Generates a formfeed after each listed function.

SYNTAX

```
-F
```

DESCRIPTION

Use `-F` to include a formfeed after each function in the listing.

-g

Enables global type check.

SYNTAX

```
-g[A][O]
```

DESCRIPTION

There is a class of conditions in the source that indicate possible programming faults but which by default the compiler and linker ignore.

The `-g` option causes the compiler to issue warning messages for these conditions, and also to include type information in the object file so that the linker will warn of them. The conditions 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.

This includes many of the conditions which on other C Compilers can be detected only by using a separate lint utility.

The -g option does not increase the size of the final code but does increase the compilation and (unless the 0 modifier is used) link times and object module size.

The A modifier enables warnings of the old-style K&R functions.

The 0 modifier inhibits the inclusion of type information in the object module, and hence inhibits type checking by the linker. Hence -g0 does not increase the object module size or link time.

Note that objects in modules compiled without type information (that is, compiled without -g[A] or with -g0[A]) are considered as totally typeless by the linker. This means that 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.

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
```

-G

Opens the standard input as source.

SYNTAX

```
-G
```

DESCRIPTION

By default, the source is read from the source file of the specified name. Use `-G` to read the source directly from the standard input stream, normally the keyboard. The source filename is set to `stdin.c`.

-H

Sets the object module name.

SYNTAX

```
-Hname
```

DESCRIPTION

By default, the internal name of the object module is the source leafname. If several modules have the same source leafname, the identical object module names causes a duplicate modules error from the linker.

This can arise, for example, when the source files are generated by a compiler pre-processor.

Use `-H` to specify an alternative object module name, to overcome this problem.

-i

Lists included files.

SYNTAX

```
-i
```

DESCRIPTION

Use the `-i` option to list `#include` files. By default they are not listed.

-I

Adds an include file search prefix.

SYNTAX

```
-Iprefix
```

DESCRIPTION

The compiler performs the following search sequence for each include file enclosed in angle brackets in a directive such as:

```
#include <file>
```

- The filename prefixed by the argument of each successive `-I` option if any.

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

In addition, if the filename is enclosed in double quotes, as in

```
#include "file"
```

the compiler first searches the filename prefixed by the source file path.

Use the `-I` option, followed immediately by a path specification, to direct the compiler to search for include files on that path.

There is no limit to the number of `-I` options on a single command line.

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.

-K

Enables C++ comments.

SYNTAX

```
-K
```

DESCRIPTION

C++ style comments are introduced by // and extend to the end of the line. By default, C++ style comments are not accepted. Use the `-K` option to allow them to be accepted.

-l

Generates a listing.

SYNTAX

```
-l file
```

DESCRIPTION

By default, the compiler does not generate a listing. Use the `-l` option to generate a listing 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, `.lst` is used.

The `-l` and `-L` options may not be used at the same time.

-L

Generates a listing.

SYNTAX

```
-Lprefix
```

DESCRIPTION

By default, the compiler does not generate a listing. The `-L` option generates a listing to a file with the same name as the source leafname but with the extension `.lst`.

The `-L` option may be followed by a prefix, which the compiler adds to the filename. This allows the user to redirect the listing to a different directory.

The `-l` and `-L` options may not be used at the same time.

-o

Specifies object filename.

SYNTAX

```
-o file
```

DESCRIPTION

Without the `-o` option, the compiler stores the object code in a file whose name is:

- The prefix specified by `-O`.
- The leafname of the source.
- A target-specific object code extension.

The `-o` option sets an entire alternative filename consisting of an optional pathname, obligatory leafname and optional extension. It allows the object code to be directed to a different file.

The `-o` and `-O` options may not be used at the same time.

-O

Specifies object filename.

SYNTAX

```
-Oprefix
```

DESCRIPTION

By default the compiler stores the object code in a file whose name is the leafname of the source plus a target-specific object code extension.

Use -O to specify a prefix which the compiler adds to the leafname, allowing the object code to redirected to an alternative directory.

The -o and -O options may not be used at the same time.

-p

Formats listing into pages.

SYNTAX

-pn

DESCRIPTION

By default, the listing is not divided into pages. Use -p followed by the number of lines per page in the range 10 to 150 to divide the listing into pages of this size.

-P

Generates PROMable code.

SYNTAX

-P

DESCRIPTION

By default, the compiler places initialized statically allocated objects in the program memory segment, and hence if the program is placed in PROM, the program cannot write to them.

Use the `-P` option to make it possible for a PROMed program to write to initialized statically allocated objects. `-P` causes the run-time system to copy initialized statically allocated objects from PROM into RAM upon start-up.

Note that `-P` is not required to enable writing to non-initialized statically allocated objects. This is because the compiler assumes that statically allocated objects that are not initialized will be written to, and hence automatically places them in RAM.

-q

Puts mnemonics in the listing.

SYNTAX

```
-q
```

DESCRIPTION

By default the compiler does not include the generated assembly lines in the compilation listing. Use `-q` to include assembly lines in the compilation listing, as an aid to debugging. See also the options `-a` and `-A`.

-r

Generates debug information.

SYNTAX

```
-r[012][i][n]
```

DESCRIPTION

By default, the object modules do not contain the additional information required by C-SPY or other symbolic debuggers. Use `-r` to include this additional information in the object code, so a debugger can be used on the module.

For the option to use to suit C-SPY see the *Using C-SPY* guide.

The following table describes the effect of the modifiers:

Modifier	*What it means*
`0`, `1`, `2`	Support different debugger hardware. For source code debuggers this information should be specified in the appropriate debugger manual. For debuggers that do not support C source line display the default (0) is sufficient.
`i`	`#include` file information will be added to the object file. Note that this is usually of little interest unless include files contain function definitions (not just declarations). Also note that C statements in `#include` files are practically non-debuggable with debuggers other than C-SPY. A side-effect is that source line records will contain the global (= total) line count which can affect source line displays in some debuggers.
`n`	Suppresses the generation of C source lines in the object file (which is only required by C-SPY and other debuggers based on the IAR debug format).

For most other debuggers that do not include specific information on how to use IAR C Compilers, `-rn` should be specified. Do not use `-r` without `n` unless specifically required, since this increases the memory requirement considerably.

Note that global optimization activated by the `-z` or `-s` options may invalidate source line information (due to statement combinations and rearrangements performed by the compiler) and that this can affect source code displays during program stepping. Also note that the `-r` option generates slightly more target code and includes type information as if `-g` had been used.

-R

Sets the code segment name.

SYNTAX

```
-Rname
```

DESCRIPTION

By default, the compiler places executable code in a segment named `CODE`, which by default the linker places at a variable address. Use `-R` to place the code in a specific segment with a unique name chosen by the user. This then allows the user to specify to the linker a fixed address for this particular segment.

-S

Optimizes for speed.

SYNTAX

```
-s[0-9]
```

DESCRIPTION

The argument sets the level of optimization:

Value	*Level*
0	No optimization.
1–3	Fully debuggable.
4–6	Some constructs not debuggable.
7–9	Full optimization.

-S

Sets silent operation of compiler.

SYNTAX

```
-S
```

DESCRIPTION

By default the compiler issues introductory messages and a final statistics report. Use `-S` to inhibit these messages.

Note that error and warning messages are shown.

-t

Sets the tab spacing.

SYNTAX

```
-tn
```

DESCRIPTION

By default, the listing is formatted with a tab spacing of 8 characters. Use `-t` to set the spacing of the tab characters to between 2 and 9 characters (default 8).

-T

Lists active lines only.

SYNTAX

```
-T
```

DESCRIPTION

By default, inactive source lines, such as those in false `#if` structures, are listed. Use `-T` to list active lines only.

-U

Undefines a symbol.

SYNTAX

```
-Usymb
```

DESCRIPTION

`-Usymb` is equivalent to:

```
#undef symb
```

By default, the compiler has the following pre-defined symbols:

Symbol	*Value*
`__IAR_SYSTEMS_ICC`	1
`__STDC__`	1
`__VER__`	Compiler version number.
`__TID__`	Target-IDENT.
`__FILE__`	Current source filename.
`__LINE__`	Current source line number.
`__TIME__`	Current time in `hh:mm:ss` format.
`__DATE__`	Current date in `Mmm dd yyyy` format.

The `-U` option can be used to switch off any of these symbols, to resolve a conflict with any user-defined symbol of the same name.

-w

Disables warnings.

SYNTAX

```
-w
```

DESCRIPTION

By default, the compiler issues standard warning messages, and any additional warning messages enabled with `-g`. Use `-w` to inhibit all warning messages.

-X

Generates a cross-reference list.

SYNTAX

```
-x[D][F][T][2]
```

DESCRIPTION

By default the compiler does not include global symbols in the listing. The `-x` option with no argument list adds a list of all global symbols and their meanings at the end of the compilation listing. This includes all variable objects and all referenced functions, `#define` statements, `enum` statements, and `typedef` statements.

To include additional information, follow `-x` by one or more of the following:

Argument	*Information*
`D`	Unreferenced `#define` symbols.
`F`	Unreferenced function declarations.
`T`	Unreferenced `enum` constants and `typedefs`.
`2`	Dual line spacing between symbol entries.

-X

Describes C declarations.

SYNTAX

```
-X
```

DESCRIPTION

Use `-X` to display a readable description of all the C declarations in the file.

EXAMPLES

For the declaration:

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

the following output will be produced:

```
Identifier: signal
storage class: extern
  prototyped non_banked function returning
    xxx - non_banked code pointer to
      prototyped non_banked function returning
        xxx - void
      and having following parameter(s):
        storage class: auto
        xxx - int
  and having following parameter(s):
    storage class: auto
    xxx - int
    storage class: auto
    xxx - non_banked code pointer to
      non_banked function returning
        xxx - void
```

-y

Initializes strings as variables.

SYNTAX

```
-y
```

DESCRIPTION

By default C string literals are assumed to be read-only. Use `-y` to generate strings as initialized variables. However, arrays initialized with strings (ie `char c[] = "string"`) are always treated as ordinary initialized variables.

-Z

Optimizes for size.

SYNTAX

`-z[0-9]`

DESCRIPTION

The argument sets the level of optimization:

Value	*Level*
0	No optimization.
1–3	Fully debuggable.
4–6	Some constructs not debuggable.
7–9	Full optimization.

See the file `GLOBAL.DOC` for additional information.

GENERAL C LANGUAGE EXTENSIONS

INTRODUCTION

The IAR C Compiler supports a number of extensions to the C language. The majority are specific to the target processor, and are therefore documented in the chapter *Language extensions*. The remainder are common to all targets and hence are documented here.

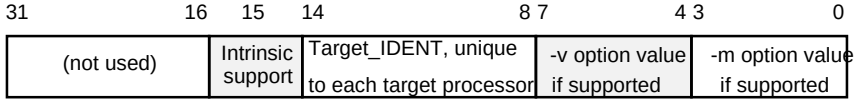
COMPILER VERSION

The macro `__VER__` returns an integer constant containing the compiler version number in decimal format.

For example, for version 2.34E the value of `__VER__` is 234.

TARGET IDENTIFICATION

The macro `__TID__` returns a long integer constant containing a target identifier and related information:



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`.

ARGUMENT TYPE

`_argt$` is a unary operator with the same syntax and argument as `sizeof`. It returns a normalized value describing the type of the argument:

Result	*Type*
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.

For an example of the use of `_argt$`, see the file `stdarg.h`.

FUNCTION PARAMETERS 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:

Offset	*Contents*
`0`	Parameter 1 type in `_argt$` format.
`1`	Parameter 1 size in bytes.
`2`	Parameter 2 type in `_argt$` format.
`3`	Parameter 2 size in bytes.
`2n-2`	Parameter `n` type in `_argt$` format.
`2n-1`	Parameter `n` size in bytes.
`2n`	`\0`

Sizes greater than 127 are reported as 127.

`_args$` may be used only inside function definitions. For an example of the use of `_args$`, see the file `stdarg.h`.

$ 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.

GENERAL C LIBRARY DEFINITIONS

INTRODUCTION

The ICC C Compiler package 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 use in user programs. These are documented in this chapter.
- `CSTARTUP`, the single program module containing the start-up code.
- Intrinsic functions, used only by the compiler, to perform low-level operations which cannot be performed by in-line code. Intrinsic functions have names beginning with ? to distinguish them from other functions. Since they are not to be used in application programs, they are not documented.

LIBRARY OBJECT FILES

For each combination of configuration and mode, there is a single library object file containing all the library definitions. 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. For many of these, the source is optionally available. The remainder are I/O-oriented routines (such as `putchar` and `getchar`) that you may need to customize for your target application. For these, the source is supplied as part of the standard installation.

The library object files are supplied having been compiled with the global type check option on (`-gA`).

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 the user 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*.

All library functions are concurrently reusable (reentrant) where stated.

CHARACTER HANDLING – ctype.h

`isalnum`	`int isalnum(int c)`	Letter or digit equality.
`isalpha`	`int isalpha(int c)`	Letter equality.
`iscntrl`	`int iscntrl(int c)`	Control code equality.
`isdigit`	`int isdigit(int c)`	Digit equality.
`isgraph`	`int isgraph(int c)`	Printable non-space character equality.
`islower`	`int islower(int c)`	Lower case equality.
`isprint`	`int isprint(int c)`	Printable character equality.
`ispunct`	`int ispunct(int c)`	Punctuation character equality.
`isspace`	`int isspace (int c)`	White-space character equality.

isupper	int isupper(int c)	Upper case equality.
isxdigit	int isxdigit(int c)	Hex digit equality.
tolower	int tolower(int c)	Converts to lower case.
toupper	int toupper(int c)	Converts to upper case.

LOW-LEVEL ROUTINES – icclbutl.h

_formatted_read		Reads formatted data.
	int _formatted_read (const char **line, const char **format, va_list ap)	
_formatted_write		Formats and writes data.
	int _formatted_write (const char* format, void outputf (char, void *), void *sp, va_list ap)	
_medium	int _formatted_read (const char **line, const char **format, va_list ap)	Reads formatted data _read excluding floating-point numbers.
_medium _write	int _formatted_write (const char* format, void outputf (char, void *), void *sp, va_list ap)	Writes formatted data excluding floating-point numbers.
_small	int _formatted_write (const char* format, void outputf (char, void *), void *sp, va_list ap)	Small formatted data _write write routine.

MATHEMATICS – math.h

acos	double acos(double arg)	Arc cosine.
asin	double asin(double arg)	Arc sine.
atan	double atan(double arg)	Arc tangent.

`atan2`	`double atan2(double arg1, double arg2)`	Arc tangent with quadrant.
`ceil`	`double ceil(double arg)`	Smallest integer greater than or equal to `arg`.
`cos`	`double cos(double arg)`	Cosine.
`cosh`	`double cosh(double arg)`	Hyperbolic cosine.
`exp`	`double exp(double arg)`	Exponential.
`fabs`	`double fabs(double arg)`	Double-precision floating-point absolute.
`floor`	`double floor(double arg)`	Largest integer less than or equal.
`fmod`	`double fmod(double arg1, double arg2)`	Floating-point remainder.
`frexp`	`double frexp(double arg1, int *arg2)`	Splits a floating-point number into two parts.
`ldexp`	`double ldexp(double arg1, int arg2)`	Multiply by power of two.
`log`	`double log(double arg)`	Natural logarithm.
`log10`	`double log10(double arg)`	Base-10 logarithm.
`modf`	`double modf(double value, double *iptr)`	Fractional and integer parts.
`pow`	`double pow(double arg1, double arg2)`	Raises to the power.
`sin`	`double sin(double arg)`	Sine.
`sinh`	`double sinh(double arg)`	Hyperbolic sine.
`sqrt`	`double sqrt(double arg)`	Square root.
`tan`	`double tan(double x)`	Tangent.
`tanh`	`double tanh(double arg)`	Hyperbolic tangent.

NON-LOCAL JUMPS – setjmp.h

longjmp	void longjmp(jmp_buf env, int val)	Long jump.
setjmp	int setjmp(jmp_buf env)	Sets jump.

VARIABLE ARGUMENTS – stdarg.h

va_arg	type va_arg(va_list ap, mode)	Next argument in function call.
va_end	void va_end(va_list ap)	Ends reading function call arguments.
va_list	char *va_list[1]	Argument list type.
va_start	void va_start(va_list ap, parmN)	Starts reading function call arguments.

INPUT/OUTPUT – stdio.h

getchar	int getchar(void)	Gets character.
gets	char *gets(char *s)	Gets string.
printf	int printf(const char *format, ...)	Writes formatted data.
putchar	int putchar(int value)	Puts character.
puts	int puts(const char *s)	Puts string.
scanf	int scanf(const char *format, ...)	Reads formatted data.
sprintf	int sprintf(char *s, const char *format,)	Writes formatted data to a string.
sscanf	int sscanf(const char *s, const char *format, ...)	Reads formatted data from a string.

GENERAL UTILITIES – stdlib.h

`abort`	`void abort(void)`	Terminates the program abnormally.
`abs`	`int abs(int j)`	Absolute value.
`atof`	`double atof(const char *nptr)`	Converts ASCII to `double`.
`atoi`	`int atoi(const char *nptr)`	Converts ASCII to `int`.
`atol`	`long atol(const char *nptr)`	Converts ASCII to `long int`.
`calloc`	`void *calloc(size_t nelem, size_t elsize)`	Allocates memory for an array of objects.
`div`	`div_t div(int numer, int denom)`	Divide.
`exit`	`void exit(int status)`	Terminates the program.
`free`	`void free(void *ptr)`	Frees memory.
`labs`	`long int labs(long int j)`	Long absolute.
`ldiv`	`ldiv_t ldiv(long int numer, long int denom)`	Long division.
`malloc`	`void *malloc(size_t size)`	Allocates memory.
`rand`	`int rand(void)`	Random number.
`realloc`	`void *realloc(void *ptr, size_t size)`	Reallocates memory.
`srand`	`void srand(unsigned int seed)`	Sets random number sequence.
`strtod`	`double strtod(const char *nptr, char **endptr)`	Converts a string to `double`.
`strtol`	`long int strtol(const char *nptr, char **endptr, int base)`	Converts a string to a long integer.

strtoul	unsigned long int strtoul (const char *nptr, char **endptr, base int)	Converts a string to an unsigned long integer.

STRING HANDLING – string.h

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

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).

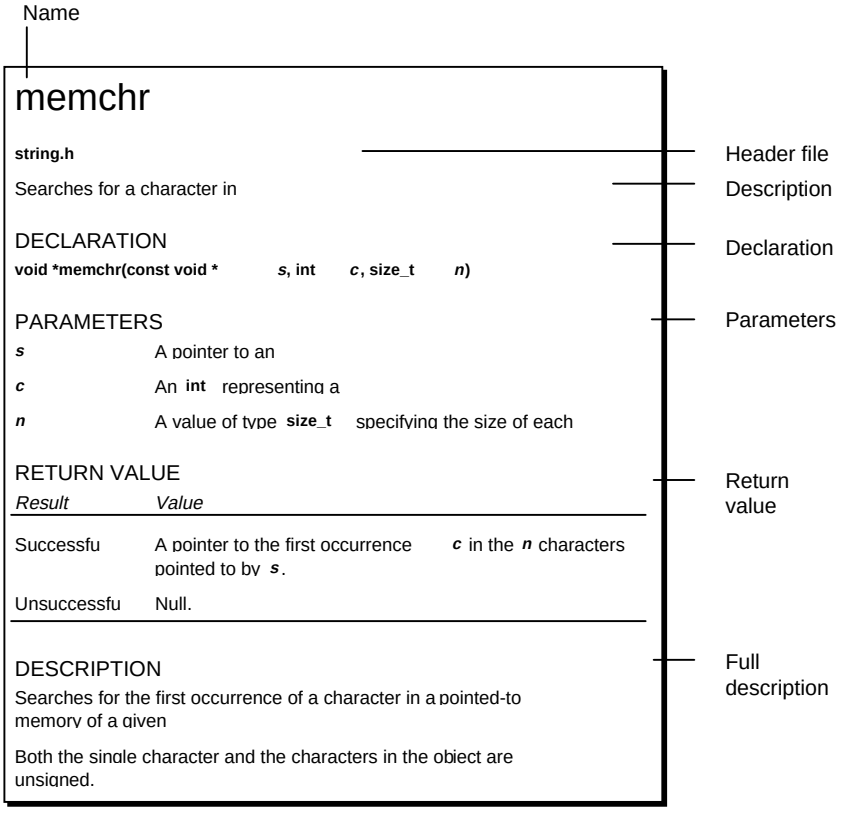
ASSERT – assert.h

`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:



NAME

The function name.

The function name is followed by the function header filename, and a brief description 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.

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.

DECLARATION

```
int 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.

DECLARATION

```
double 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

calloc

`stdlib.h`

Allocates memory for an array of objects.

DECLARATION

```
void *calloc(size_t nelem, size_t elsize)
```

PARAMETERS

`nelem`	The number of objects.
`elsize`	A value of type `size_t` specifying the size of each object.

RETURN VALUE

Result	*Value*
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

`arg`	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

Terminate 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`.

fabs

`math.h`

Double-precision floating-point absolute.

DECLARATION

```
double 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.

DECLARATION

```
void 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.

DECLARATION

```
double 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:

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

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.

The user must 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

Result	*Value*
Successful	A pointer equal to `s`.
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.

iscntrl

`ctype.h`

Control code equality.

DECLARATION

```
int iscntrl(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

`ctype.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

`ctype.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:

Character	*Symbol*
Space	' '
Formfeed	\f
New line	\n
Carriage return	\r
Horizontal tab	\t
Vertical tab	\v

isupper

`ctype.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

Test 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.

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

Result	*Value*
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 default heap size.

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

Result	*Value*
Successful	A pointer to the first occurrence of `c` in the `n` characters pointed to by `s`.
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

`s1`	A pointer to the first object.
`s2`	A pointer to the second object.
`n`	A value of type `size_t` 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`:

Return value	*Meaning*
`>0`	`s1 < s2`
`=0`	`s1 = s2`
`<0`	`s1 < s2`

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 VALUE

The 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

Result	*Value*
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* in the target-specific section.

`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:

Flag	*Effect*	
-	Left adjusted field.	
+	Signed values will always begin with plus or minus sign.	
space	Values will always begin with minus or space.	
#	Alternate form:	
	specifier	*effect*
	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:

Length_modifier	*Use*
`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:

Conversion	*Result*
`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.

Conversion	*Result*
G	Double value in the style of F or E, whichever is the more appropriate.
C	Single character constant.
s	String constant.
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; _ represents space:

Statement	*Output*	*Number of characters output*
printf("%c",p[1])	B	1
printf("%d",i)	6	1
printf("%3d",i)	__6	3
printf("%.3d",i)	006	3
printf("%-10.3d",i)	006_____	10
printf("%10.3d",i)	_____006	10
printf("Value=%+3d",i)	Value=_+6	9
printf("%10.*d",i,j)	___-000006	10
printf("String=[%s]",p)	String=[ABC]	12
printf("Value=%lX",l)	Value=186A0	11
printf("%f",f1)	0.000000	8
printf("%f",f2)	750000.000000	13
printf("%e",f1)	1.000000e-07	12
printf("%16e",d)	____2.200000e+00	16
printf("%.4e",d)	2.2000e+00	10
printf("%g",f1)	1e-07	5
printf("%g",f2)	750000	6
printf("%g",d)	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

Result	*Value*
Successful	`value`.
Unsuccessful	The `EOF` macro.

DESCRIPTION

Writes a character to standard output.

The user must 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

Result	*Value*
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.

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

Result	*Value*
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

Result	*Value*
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 the chapter *Configuration* in the target-specific section.

`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:

Length modifier	*Before*	*Meaning*
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:

Conversion	*Meaning*
`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.
`e E g G`	Floating-point constant (equivalent to `f`).
`s`	Character string.
`c`	One or `field_width` 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 `]`. For example, `[abc]` means `a`, `b`, or `c`.
`[]`	Any number of characters matching `]` or any of the characters before the further, terminating `]`. For example, `[]abc]` means `]`, `a`, `b`, or `c`.
`[^`	Any number of characters not matching any of the characters before the terminating `]`. For example, `[^abc]` means not `a`, `b`, or `c`.

Conversion	*Meaning*
`[^]`	Any number of characters not matching `]` or any of the characters before the further, terminating `]`. For example, `[^]abc]` means not `]`, `a`, `b`, or `c`.
`%`	`%` character.

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

`scanf` indirectly calls `getchar`, 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 jump.

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

`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.

sprintf

stdio.h

Writes formatted data to a string.

DECLARATION

```
int sprintf(char *s, const char *format, …)
```

PARAMETERS

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

RETURN VALUE

Result	*Value*
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.

sprintf 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 the chapter *Configuration* in the target-specific section.

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

Result	*Value*
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* in the target-specific section.

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:

Return value	*Meaning*
`>0`	`s1 < s2`
`=0`	`s1 = s2`
`<0`	`s1 < s2`

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:

Return value	*Meaning*
`>0`	`s1 < s2`
`=0`	`s1 = s2`
`<0`	`s1 < s2`

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.

DECLARATION

```
size_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.

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:

Return value	*Meaning*
`>0`	`s1 < s2`
`=0`	`s1 = s2`
`<0`	`s1 < s2`

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

Result	*Value*
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

Result	*Value*
Successful	A pointer to the first occurrence in the string pointed to by `s1` of the sequence of characters (excluding the terminating null character) in the string pointed to by `s2`.
Unsuccessful	Null if the string was not found. `s1` if `s2` 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

`nptr`	A pointer to a string.
`endptr`	A pointer to a pointer to a string.

RETURN VALUE

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

DESCRIPTION

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

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

Result	*Value*
Successful	The `long int` result of converting the ASCII representation of an integer constant in the string pointed to by `nptr`, leaving `endptr` pointing to the first character after the constant.
Unsuccessful	Zero, leaving `endptr` 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

Result	*Value*
Successful	The unsigned long int result of converting the ASCII representation of an integer constant in the string pointed to by nptr, leaving endptr pointing to the first character after the constant.
Unsuccessful	Zero, leaving endptr 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.

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.

DECLARATION

```
type 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

icclbutl.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

`icclbutl.h`

Formats and writes data.

DECLARATION

```
int _formatted_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

Formats write data. This function is the basic formatter of `printf` and `sprintf`, but through its universal interface can easily be adapted by the user 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* in the target-specific section.

`_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

icclbutl.h

Reads formatted data excluding floating-point numbers.

DECLARATION

```
int _medium_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

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

`icclbutl.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

`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

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

`icclbutl.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

A small version of `_formatted_write` which is about a quarter of the size, and uses only about 15 bytes of RAM.

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`.

K&R AND ANSI C LANGUAGE DEFINITIONS

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*. The IAR C Compiler is fully compatible with this definition.

- 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 C Compiler closely 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 that are familiar with K&R C but would like to use the new ANSI facilities.

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:

K&R	*ANSI*
`long int g(s)` `char * s;`	`long int g(char * s);`
`{`	`{`

The arguments of ANSI-type functions are always type-checked. The IAR C Compiler checks the arguments of K&R-type functions only if the `-g` option is used.

FUNCTION DECLARATIONS

In K&R C, function declarations do not include parameters. In ANSI C they do. For example:

Type	*Example*
K&R	`extern int f();`
ANSI (named form)	`extern int(long int val);`
ANSI (unnamed form)	`extern int(long int);`

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 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
                        struct s */
a = f(b);                           /* call it */
```

To further increase the usability of structures, 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 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:

Module #1	*Module #2*	*Module #3*
`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 produced 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.

In addition to these general error and warning messages, any target-specific error and warning messages are documented in the chapter *Diagnostics.*

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.

The ICC C Compiler is more strict on compatibility issues than many other C compilers. In particular pointers and integers are considered as incompatible when not explicitly casted. Compilation error messages are described in *Compilation error messages* in this chapter.

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 warning messages are described in *Compilation warning messages* in this chapter.

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 `-q`, `-x`, `-A`, `-P`, and `-r` (not `-rn`) switches cause the compiler to use more memory.

Also, see the chapter *Getting Started*, 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 diskette with the program that generated the internal error.

COMPILATION ERROR MESSAGES

No	*Error message*	*Suggestion*
0	`Invalid syntax`	The compiler could not decode the statement or declaration.
1	`Too deep #include nesting (max is 10)`	Fatal. The compiler limit for nesting of `#include` files was exceeded. One possible cause is an inadvertently recursive `#include` file.
2	`Failed to open #include file 'name'`	Fatal. The compiler could not open an `#include` file. Possible causes are that the file does not exist in the specified directories (possibly due to a faulty `-I` prefix or `C_INCLUDE` path) or is disabled for reading.
3	`Invalid #include filename`	Fatal. The `#include` filename was invalid. Note that the `#include` filename must be written `<file>` or `"file"`.
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 `()` or `{}` nesting.

No	*Error message*	*Suggestion*
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 `0x` or `0X` 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.
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 strings was exceeded.
13	`Argument to #define too long (max is 512)`	Lines terminated by \ resulted in a `#define` line that was too long.

No	*Error message*	*Suggestion*
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.
19	`Unmatched #else, #endif or #elif`	Fatal. A `#if`, `#ifdef`, or `#ifndef` was missing.
20	`No such pre-processor command: 'name'`	# 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.

No	*Error message*	*Suggestion*
25	`Macro 'name' called with too many parameters`	Fatal. A parameterized `#define` 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 `#define` macro was called with fewer arguments than declared.
28	`Missing #endif`	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 `struct`, `union`, prototyped function definitions/declarations, or in K&R function formal parameter declarations.
30	`Identifier unexpected`	There was an invalid identifier. This could be an identifier in a type name definition like: `sizeof(int*ident);` 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.

No	*Error message*	*Suggestion*
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 `-ri` 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 nn)`	The declared width of field exceeds the size of an integer (`nn` is 16 or 32 depending on the target processor).
39	`',' or ';' expected`	There was a missing `,` or `;` 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.

No	*Error message*	*Suggestion*
43	`Object cannot be initialized`	There was an attempted initialization of `typedef` declarator or `struct` or `union` member.
44	`';' expected`	A statement or declaration needs a terminating semicolon.
45	`']' expected`	There was a bad array declaration or array expression.
46	`':' expected`	There was a missing colon after `default`, `case` label, or in ?-operator.
47	`'(' expected`	The probable cause is a misformed `for`, `if`, or `while` statement.
48	`')' expected`	The probable cause is a misformed `for`, `if`, or `while` statement or expression.
49	`',' expected`	There was an invalid declaration.
50	`'{' expected`	There was an invalid declaration or initializer.
51	`'}' expected`	There was an invalid declaration or initializer.
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.

No	*Error message*	*Suggestion*
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:	
	`int array [ 5 ] ();` `/* Invalid */` `int (*array [ 5 ]) ();` `/* Valid */`	
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: `int f(i); /* Invalid */`
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.
63	`Function returning function`	A function cannot return a function, as in: `int f()(); /* Invalid */`

No	*Error message*	*Suggestion*
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.

No	*Error message*	*Suggestion*
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`	There is a pointer to a field member of `struct` or `union`:
		`struct`
		`  {`
		`  int *f:6;  /* Invalid */`
		`  }`

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.

No	*Error message*	*Suggestion*
81	`Declaration after statement`	A declaration was found after a statement.

	This could be due to an unwanted `;` for example:
	`int i;;`
	`char c;  /* Invalid */`
	Since the second `;` is a statement it causes a declaration after a statement.	

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.

No	*Error message*	*Suggestion*
86	`Invalid formal parameter`	There is a fault with the formal parameter in a function declaration.

Possible causes are:

```
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 */
```

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:

```
int f( ...  );              /* Invalid */
int f( int, ...  );         /* Valid */
```

90	`Formal parameter identifier missing`	An identifier of a parameter was missing in the header of a prototyped function definition.

For example:

```
int f( int *p, char, float ff) /* Invalid - second
                                  parameter has no name
*/
  {
      /* function body */
  }
```

No	*Error message*	*Suggestion*
91	`Redeclared number of formal parameters`	A prototyped function was declared with a different number of parameters than the first declaration.
	For example:	

	`int f(int, char);  /* first declaration-valid */`
	`int f(int);  /* fewer parameters-invalid */`
	`int f(int, char, float);  /* more parameters-invalid */`	

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.
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.

No	*Error message*	*Suggestion*
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 argument that was not 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: `char array [ 4 ] = "abcde"; /* invalid */`
105	`Language feature not implemented: 'name'`	The compiler does not currently support the language feature used.
106	`Too many function parameters (max is 127)`	Fatal. There were too many parameters in function declaration/definition.

No	*Error message*	*Suggestion*
107	`Function parameter 'name' already declared`	A formal parameter in a function definition header was declared more than once.

For example:

```
/* K&R function */ int myfunc( i, i ) /* invalid */
int i;
  {
  }
/* Prototyped function */
int myfunc( int i, int i ) /* invalid */
  {
  }
```

108	`Function parameter 'name' declared but not found in header`	In a K&R function definition, parameter declared but not specified in the function header.

For example:

```
int myfunc( i )
int i, j /* invalid - j is not specified in the function
header   */
  {
  }
```

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.

No	*Error message*	*Suggestion*
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 was a `const`.
123	`Prototyped function argument number mismatch`	A prototyped function was called with a number of arguments different from the number declared.

No	*Error message*	*Suggestion*
124	`Unknown "struct" or "union" member: 'name'`	An attempt was made to reference a nonexistent 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.
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:

```
char (*(*p1)[8])(int);
char (*(*p2)[8])(float);

/* p1 and p2 are incompatible - the function parameters
have incompatible types */
```

The compatibility test also includes checking of array dimensions if they appear in the description of the objects pointed to, for example:

```
int (*p1)[8];
int (*p2)[9];

/* p1 and p2 are incompatible – array dimensions differ
*/
```

128	`Function argument incompatible with its declaration`	A function argument is incompatible with the argument in the declaration.

No	*Error message*	*Suggestion*
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.
133	`Constant value required`	The expression in a `case` label, `#if`, `#elif`, 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.

No	*Error message*	*Suggestion*
138	`"- pointer" expression`	The - operator may be used on pointers only if both operators are pointers, that is, `pointer - pointer`. This error means that an expression of the form `non-pointer - pointer` 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	`#error 'any message'`	Fatal. The pre-processor directive `#error` was found, notifying that something must be defined on the command line in order to compile this module.
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.

No	*Error message*	*Suggestion*
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` (enable type check) compiler 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`.
149	`Attempt to take address of "sfr" or "bit" variable`	The & operator may not be applied to variables declared as `bit` or as `sfr`.

No	*Error message*	*Suggestion*
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`	See *Diagnostics*.

COMPILATION WARNING MESSAGES

No	*Warning message*	*Suggestion*
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.

No	*Warning message*	*Suggestion*
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, bit-fields 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.

No	*Warning message*	*Suggestion*
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.
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:	
	`#define mac(p1) #p1 /* Becomes "p1" */` `#define mac(p1,p2) p1+p2##add_this /* Merged p2 */`	
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.

No	*Warning message*	*Suggestion*
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.
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;                  /* Never executed */
```

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;                  /* Never executed */
```

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.

No	*Warning message*	*Suggestion*
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`.
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`	A K&R function was called with changing types of parameters.
	For example:	
	`myfunc ( 34 );  /* int argument */` `myfunc( 34.6 );  /* float argument */`	
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`.

No	*Warning message*	*Suggestion*
30	`Attribute ignored`	Since `const` or `volatile` 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 `const` or `volatile`.

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
```

No	*Warning messages*	*Suggestion*
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 pointer
pointer 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.

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z