IAR C COMPILER FOR THE Z80/64180

Second edition: March 1997

Part number: ICCZ80-2

ABOUT THIS GUIDE

This guide describes how to install and use the IAR C Compiler for the Zilog Z80, Zilog Z80180 series (referred to as Z8018X), and Hitachi 64180 microprocessors.

This guide is divided into two parts: the first part, *IAR Z80/64180 C Compiler,* describes those aspects of the C compiler that are specific to the Z80/64180. The second part, *IAR C Compiler - General Features,* describes features common to all IAR C Compilers.

IAR Z80/64180 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 Z80/64180 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.

Language extensions describes the extended keywords, #pragma keywords, and intrinsic functions specific to the Z80/64180 C Compiler.

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

#pragma directives reference gives information about the #pragma keyword extensions for the Z80/64180.

Intrinsic junction reference gives information about the intrinsic functions for the Z80/64180.

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

Z80 command line options describes the additional command line options in the Z80/64180 C Compiler.

Finally, *Z80 diagnostics* lists the Z80-specific warning and error messages.

IAR C COMPILER - GENERAL FEATURES

This part consists of the following chapters:

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

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

C library junctions 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 Z80, Z8018X or 64180 processor.
- The target 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 CProgrammingLanguageby* Kemighan 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.
*[opt i*on]	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 Kemighan and Richie.

CONTENTS

IAR Z80/64180 C COMPILER

IAR Z80/64180
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 Z80, Z8018X, and 64180 microprocessors offers the standard features of the C language, plus many extensions designed to take advantage of specific features of the Z80/64180. The compiler is supplied with the IAR Assembler for the Z80/64180, with which it is integrated and shares linker and library manager tools.

KEY FEATURES

The IAR C Compiler for the Z80/64180 offers the following key features:

LANGUAGE FACILITIES

- Conformance to K&R and ANSI specifications.
- Standard library of functions applicable to embedded systems, with sources included.
- IEEE-compatible floating-point arithmetic.
- Powerful extensions for features specific to Z80, Z8018X, and 64180, 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 significant 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

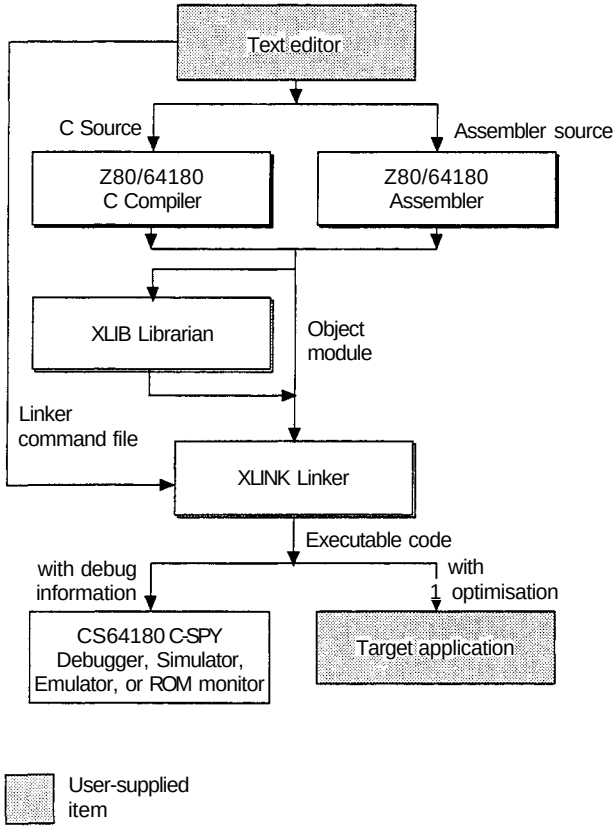
- Selectable optimization levels for code speed and size.
- Comprehensive output options, including relocatable binary, ASM, ASM + C,XREF,etc.
- Easy-to-understand error and warning messages.
- Compatibility with C-SPY high-level debugger, simulator and emulator driver.
- Support for over 20 emulator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

TARGET SUPPORT

- 64 Kbyte and banked memory models.
- SFR type for I/O, with header files. Zilog Z80180, Z80181, Z80182, and Hitachi HD64180.
- Interrupt functions requiring no assembly language.
- A #p r a g m a directive to maintain portability while using Z80 extensions.

DEVELOPMENT SYSTEM STRUCTURE

The following diagram shows how the IAR C Compiler is used as part of a complete IAR 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 maybe 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

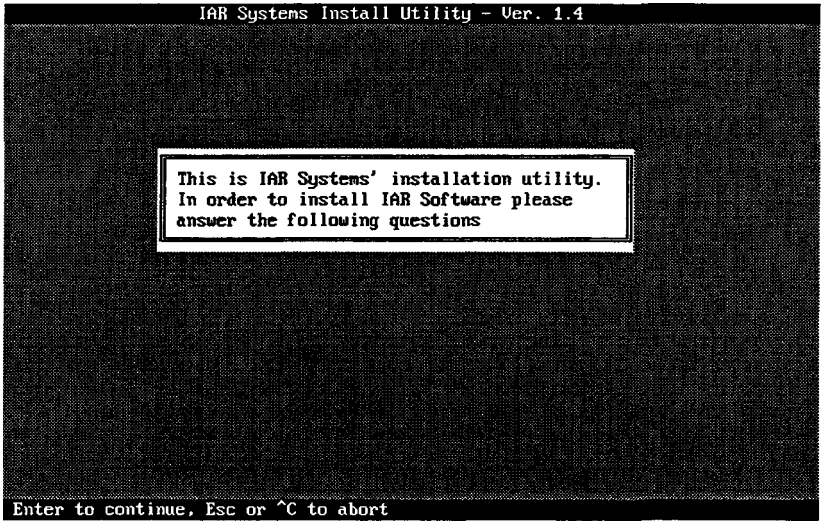
This chapter shows you how to install all files from the installation disks supplied, describes the installed files themselves, and lists the file extensions used by the system.

INSTALLATION UNDER MS-DOS

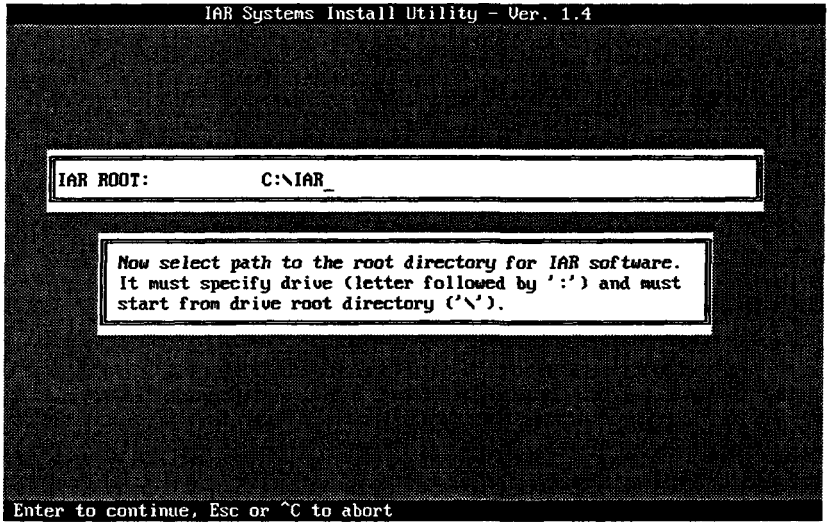
- Ensure your system has MS-DOS 4.01 or higher.
- Insert the installation disk into the floppy disk drive and type:

  a:\insta11 0

The startup screen is displayed:



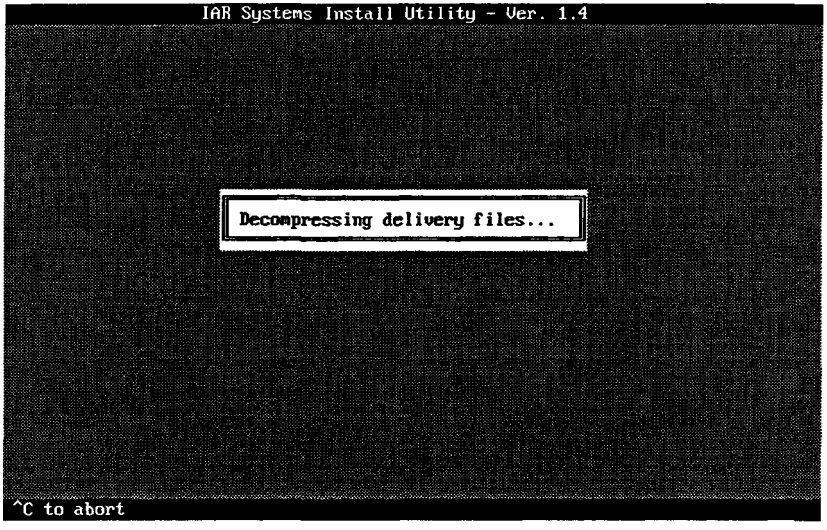
- Press Q- You will then be prompted to enter the path for installing the IAR subdirectories and files:



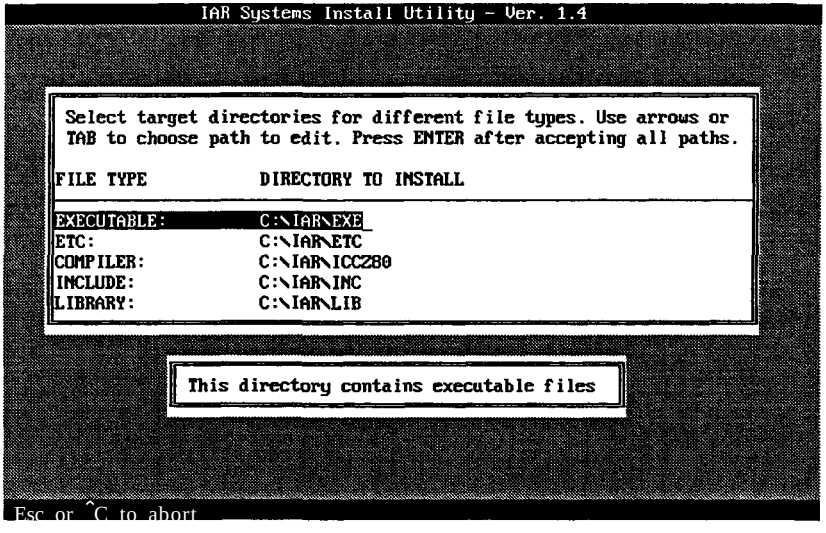
By default the files are installed in c: \ i a r.

- Edit the path, or press Q 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 subdirectory into which the files will be installed, similar to this:



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 Q 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 confinnation 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 maybe 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 UNLX

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:

File	*Description*
C: \i a r\etc\newcl i b. doc	Documentation of additional C library functions.
C: \i ar\i ccz80\i ccz80. doc	General information about the C compiler.
C: \i ar\i ccz80\i ccz80 .his	Product history: bug fixes and added features.

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 subdirectories 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*
Z80/64180 C Compiler	iccz80	This guide
Z80/64180 Assembler	a z 8 0	*LAR Assembler, Linker, & Librarian for the Z80/64180 Series*
IAR Linker	xl i n k	*IAR Assembler, Linker, & Librarian for the Z80/64180 Series*
IAR Librarian	xl i b	IAR *Assembler, Linker, & Librarian for the Z80/64180 Series*
C-SPY Z80/64180 Debugger	cs 64180	*IAR Using C-SPY Guide*

Note that C-SPY is supplied separately. For further details please refer to the documentation that comes with C-SPY.

The installation procedure, described above, creates several directories and installs in them a number of files. The following sections give a detailed description of these files.

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 subdirectory for command files. This allows the user to issue an IAR command from any directory.

The c:\iar\exe subdirectory contains the following MS-DOS executable program files:

Name	*Function*
az80.exe	Z80/64180 Assembler; see the *IAR Z80/64180 Assembler* guide.
xlib.exe	IAR Library Manager; see the *IAR Z80/64180 Assembler* guide.
xlink.exe	IAR Linker; see the *IAR Z80/64180 Assembler* guide.
pminfo.exe	IAR Protected Mode Analyzer; see the IAR *Z80/64180 Assembler* guide.
bnksetup.exe	64180/Z818Xbank setup utility; see *Changing the banked memory specification,* page 1-65.
rminfo.exe	DOS/16M real mode information program; see *Checking extended memory,* page 1-20.
iccz80.exe	Z80/64180 C Compiler; see *Z80 command line options,* page 1-137.

If you have installed the C-SPY Debugger, this subdirectory will also contain cs64180 .exe, the C-SPY Z80/64180 Debugger; see the *Using C-SPY* guide.

MISCELLANEOUS FILES

The c:\iar\etc subdirectory holds miscellaneous files such as read-me files and example sources.

It contains the following files:

Name	*Function*
emu! a tor. doc	Documentation on supported emulators.
xlink.doc	Additional information on the linker, XLINK.
n ewe 1 i b. d o c	Information on additional C library functions.
i n tw r i. c	Source of the minimal p r i n t f implementation, as an example. See *C library functions reference* in the JAR *C Compiler - General Features* guide.
sprintf.c	Source of the standard sprint f, as an example of va_a rg use. See *C library functions reference* in the *IAR C Compiler - General Features* guide.
printf.c	Source of the standard printf, as an example of va_a rg use. See *C library functions reference* in the *IAR C Compiler - General Features* guide.
frmrd.c	Source for formatted read.
f mw r i . c	Source for formatted write.
a z 8 0. s 01	Assembler main source file, invoking a s z 8 0 i n s and asz80opr below.
az80ins.s01	Assembler include source file testing assembly of all standard Z80 instructions.
az80opr. sOl	Assembler i ncl ude source file testing assembly of all assembler operators.
az80ext.s01	Assembler include source file testing assembly of all extended Z80 instructions.
sieve.c	Source of the sieve example program.

SOURCE FILES

The c:\iar\i ccz80 subdirectory holds source files for configuration to the target environment and program requirements, as described in *Configuration*, page 1-59.

This subdirectory contains the following configuration starting-point files:

Name	*Function*
iccz80.doc	Additional information about the C compiler.
lnkz80.xcl	Linker command file for Z80 large memory model.
lnkz80b.xcl	Linker command file for Z80 banked memory model.
lnk64.xcl	Linker command file for 64180 and Z8018X large memory model.
lnk64b.xcl	Linker command file for 64180 and Z8018X banked memory model.
putchar.c	Source of putchar.
getchar.c	Source ofgetchar.
cstartup.sOl	The source for CSTARTUP.
108.sOl	The source for the bank switching system.
bnksetup.c	The bank setup programs.

C INCLUDE FILES

The c:\iar\inc subdirectory 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 1-17, and the installation procedure includes the i nc subdirectory 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.

This subdirectory contains the following C include files:

Name	*Function*
assert.h ctype.h errno.h float.h limits.h math.h setjmp.h stdarg.h stddef.h stdio.h stdlib.h string.h	Header files for standard C library functions; see *C library functions reference* in the *IAR C Compiler - General Features* guide.
icclbutl.h	The source header for use by printf.c; see *Miscellaneous files*, page 1-11.
iccext.h	The source header for internal library definitions, not for use by user.
intrz80.h	The source file for both Z80 and 64180 intrinsic functions.
intz80.h	Header file with defines for Z80 interrupts.

C compiler header files for I/O.

Name	*Function*
io80180.h	I/O addresses for the Z80180 processor.
io80181.h	I/O addresses for the Z80181 processor.
io80182.h	I/O addresses for the Z80182 processor.
io64180.h	I/O addresses for the 64180 processor.

Assembler header files for I/O.

Name	*Function*
i 08OI8O. i nc	I/O addresses for the Z80180 processor.
i 08OI8I. i nc	I/O addresses for the Z80181 processor.
i o80182. i nc	I/O addresses for the Z80182 processor.
io64180.inc	I/O addresses for the 64180 processor.

LIBRARY FILES

The c:\iar\1ib subdirectory holds library modules.

The linker searches for library files in the directories given in the XLINK_DFLTDIR environment variable (see *XLINK environment variables* in the JAR *Linker &. Librarian* guide), and the installation procedure includes the 1 i b subdirectory 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.

This subdirectory contains all the library modules, as follows:

Name	*Function*
c 1 z 8 0. r 01	Library file for Z80 large memory model.
clz80b.r01	Library file for Z80 banked memory model.
cl 64. rOl	Library file for Z8018X and 64180 large memory model.
cl 64b. rOl	Library file for Z8018X and 64180 banked memory model.

ASSEMBLER FILES

The c: \i a r\az80 subdirectory holds assembler-specific files; see the JAR *Z80/64180 Series Assembler* guide.

C-SPY FILES

If you have installed the C-SPY Debugger (supplied separately), there will also be a c: \ i a r \cs64180 subdirectory holding the C-SPY-specific files; see the *Using C-SPY* guide.

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
.his	ASCII documentation	-	Text editor
.exe	MS-DOS program	-	MS-DOS command
.c	C program source	Text editor	ICCZ80 Compiler
.h	C header source	Text editor	C#i include
.sOl	Asm program source	Text editor	AZ80 Assembler
.inc	Asm include source	Text editor	Asm#include
.xcl	Linker command files	Text editor	XLINK
.rOl	Object module	ICCZ80, AZ80	XLINK, XLIB
.aOl	Target program	XLINK	EPROM, C-SPY, etc.
.dOl	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.

Note that by default linker listings (maps) will have the .1st extension. This may overwrite the listing file generated by the compiler. It is recommended that you explicitly name XLINK map files, such as example.map.

USING THE C COMPILER

RUNNING THE C COMPILER

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

iccz80 *[options] [sourcefi 1 el [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.

sourcefi 1 e The name of the source file.

If no *options* or *sourcefi* 7eis given, the command displays information about the compiler, including a summary of the options and the target-specific file extensions used.

OPERATION UNDER UNLX

Filenames are case sensitive so, for example, prog ram. 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 maybe given in one of two ways:

Standard search sequence

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

<f7'7e>

For example:

#include *<incfile.h>*

The standard search sequence is as follows:

- The include filename with successive prefixes set with the -1 option if any.
- The include filename with successive prefixes set in the environment variable named C_INCLUDE if present. Multiple prefixes maybe 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    "incfiTe.h"
```

For example, with a source file named \project\prog. c, the compiler would first look for the file \project\ i ncf i 1 e. 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 . sOl 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 . rOl 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 .1st.

EXTENDED COMMAND LINE FILE

In addition to accepting options and source filenames from the command line, the compiler can accept them from an extended command line file, or from the QCCZ80 environment variable.

Extended command line files have the extension .xcl by default, and can be specified using the -f command line option. For example, to read the command line options from extend.xcl and extend2.xcl enter:

```
iccz80 -f extend -f extend2
```

The QCCZ80 environment variable can be set up using the MS-DOS set command. For example typing:

```
set QCCZ80 —z
```

at the MS-DOS prompt or including this in the autoexec.bat file will cause the compiler to optimize for size in all compilations.

CHECKING EXTENDED MEMORY

The Z80 C Compiler can take advantage of DOS/16M-compatible extended memory, and a utility pminfo is provided to test your PC's extended memory for compatibility with DOS/16M. The utility also provides information about the memory available in your system. To run the utility, type:

```
pminfo Q
```

If pminfo does not return the amount of extended memory, or if it crashes, then your memory implementation is not compatible with DOS/16M.

Use the table below for values for the D0S16M environment variable and experiment until pminfo works correctly. If you cannot get pminfo to work after making changes to D0S16M, you probably have a PC that is incompatible with the IAR Systems extended memory system.

The utility rminfo will give you additional information about your machine and its configuration.

Machine	*Setting*	*Description*
386/486 with DPMI	0	Automatically set if DPMI is active
NEC 98	1	*
PS/2	2	Automatically set
80386	3	Automatically set
80386	INBOARD	For Intel Inboard **
Fujitsu FMR-60,-70	5	*
AT&T 6300	6	**
82C30 processor	8	(Tandy for example)
80286	9	Automatically set
80286	10	Fast switching alternative **
386/486 with VCPI	11	Automatically set if VCPI is active
Zenith Z-24K	13	Older BIOS version
Hitachi B16, B32	14	*
OKI if800	15	*
IBM PS/55	16	A
286	19	Earlier version of setting 9

* You must set D0S16M for these machines.

** You must set D0S16M for these machines and specify the memory range.

For example, to set DOS/16M to work with an Intel 386 Inboard, type:

```
set D0S16M=INB0ARD
```

For settings which require a memory range to be specified, place it after the setting:

```
set D0S16M=setting @startt-end] [:size~\
```

Use decimal or hexadecimal (Ox prefix) with a trailing K or M to indicate the addresses and size (if K or M is omitted, K is assumed).

For example:

```
set D0S16M=6 @s2M-4M
set D0S16M=6 @s4M:512
set D0S16M=6 :0x100 @2M
```

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 run a program using C-SPY.
- How to use the following Z80/64180 series-specific features: #pragma directives, provided header files, and banked memory.

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

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

If you are not using C-SPY, you may still follow this tutorial by examining the list files, or by using an alternative debugger. The `.lst` and `.map` files show which areas of memory to monitor.

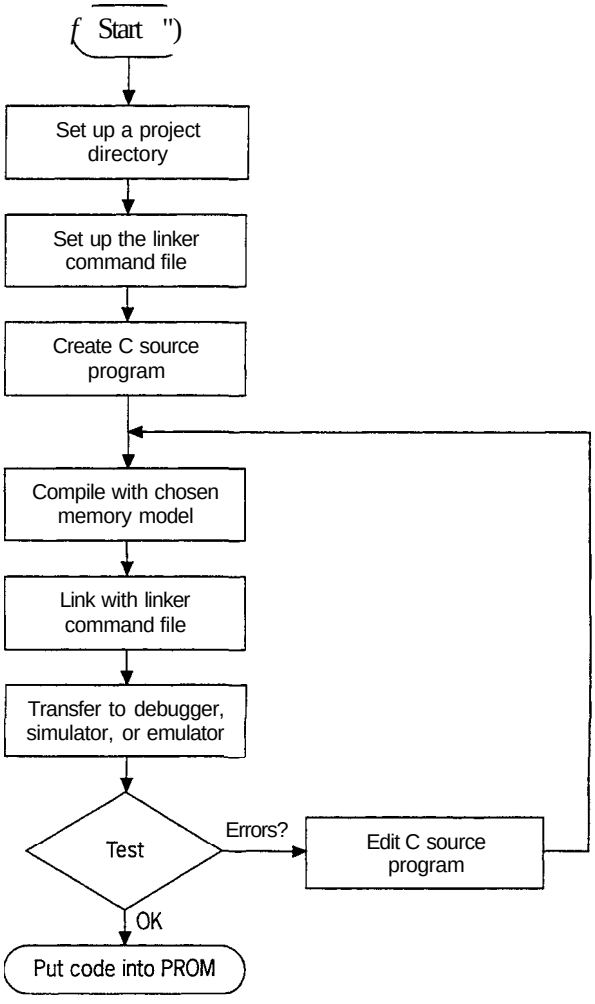
Summary of tutorial files

The following tables summarize the tutorial files used in this chapter:

File	*What it demonstrates*
tutorl	Simple C program.
tutor2	Using serial I/O.
tutor3	Interrupt handling.
tutor4a	Using banked memory.
tutor4b	Using banked memory.
tutor4c	Using banked memory.

TYPICAL DEVELOPMENT CYCLE

Development will normally follow the cycle illustrated below:



The following tutorial follows this cycle.

CREATING A PROJECT DIRECTORY

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

Create a project directory by entering the command:

```
mkdir c:\tutoria1 (P)
```

Select the project directory by entering the command:

```
cd c:\tutoria1 0
```

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\iccz80\1nk64.xcl Ink.xcl
```

This creates a copy called Ink.xcl in your project directory. The 64180 processor is used for these tutorials, but the steps for using the z80 are similar.

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 the first tutorial program, use the following locations, which are appropriate to a typical target system with 48K or ROM and 2K of RAM:

Memory	*Description*	*Address*
ROM	Code and constants	0x0000
RAM	Data variables	OxEOOO

Because we are using C-SPY instead of a real target, you could actually specify any reasonable ROM and RAM addresses and C-SPY will automatically simulate them.

- The amount of RAM required for the stack.

The tutorial program has few dynamic variables and no deep nesting of function calls, so a 256 (0x100) byte stack is ample.

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

A section of the 1 n k. xc 1 file is shown below:

```
-!  First allocate read only segments.
    0 is supposed to be start of PROM
    0-FF is left free for RST routines -!

-Z(C0DE)RC0DE,CODE,CDATAO.ZVECT,CONST.CSTR.CCSTR-100-BEFF

-!  The interrupts vectors are supposed to start at BF00
    and take 256 (FFH) bytes                              -!

-Z(C0DE)INTVEC-BF00-BFFF

-!  Then the writable segments which must be mapped to a RAM area
    C000 was here supposed to be start of RAM.
    Note: Stack size is set to 512 (200H) bytes with 'CSTACK+200    -!

-Z(DATA)DATAO.IDATAO.UDATAO.ECSTR.WCSTR.TEMP.CSTACK+200-E000-FFFF
```

The ROM area is divided into three parts: the restart vectors from 0x0000 to OxOOFF, the code and constants from 0x0100 to OxBEFF, and the interrupt vector table from OxBFOO to OxBFFF. The linker will later divide the area from 0x0100 to OxBEFF into subparts for each of the memory for the code, numerical constants, initial values, and constant strings.

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 1-63. For detailed information on choosing stack size, see *Stack size,* page 1-69.

SELECTING A LIBRARY FILE

Library selection involves a single choice:

Memory model large or banked.

See *Memory model,* page 1-61, for more details.

Our tutorial program contains only a small amount of code, and so requires only the large memory model as opposed to the banked memory model. The appropriate library file for this is cl 64180. rOl.

See *Library files,* page 1-14, 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. Alternatively, a copy is supplied in the file tutorl.c in the C:\iar\iccz80 directory:

```
#include <stdio.h>

int call_count;
unsigned char my_char;
const char con_char='a';

void do_foreground_process(void)
{
  int fp_var=l;
  call_count++;
  putchar(my_char);
}
```

```
void set_1ocal(void)
{
  int inc_var=l;
  char inc_char;
  inc_char='b';
}

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

Save the source as the file tutor1. c.

COMPILING THE PROGRAM

To compile the program, enter the command:

iccz80 -ml -r -L -q -vl tutorl Q

There are several compile options used here:

Option	*Description*
-ml	Selects the large memory model.
-r	Allows the code to be debugged using C-SPY.
-L	Creates a list file.
-q	Includes assembler code with C in the list.
-v1	Produces code for the 64180.

This creates an object module called tutorl. rOl and a list file called tutorl.1st.

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

```
\   0000              do_foreground_process:
    1           #include  <stdio.h>
    2
    3           int  can_count;
    4           unsigned  char  my_char;
    5           const  char  con_char-'a':
    6
    7           void  do_foreground_process(void)
    8            {
\   0000  CD0000            CALL     ?ENT_AUT0_DIRECT_L09
\   0003  FEFF              DEFW     -2
    9             int fp_var-l;
\   0005  DD36FE01          LD       (IX-2).l
\   0009  DD36FF00          LD       (IX-l).O
   10           call_count++;
\   000D  2A000O            LD       HL.(call_count)
\   0010  23                INC      HL
\   0011  220000            LD       (call_count).HL
   11           putchar(my_char);
\   0014  ED5B0200          LD       DE.(my_char)
\   0018  1600              LD       D.O
\   001A  CD0000            CALL     putchar
   12          }
\   001D  C30000            JP       ?LEAVE_DIRECT_L09
\   0020            set_local:
   13
   14          void  set_local(void)
   15           {
\   0020  CD0000            CALL     ?ENT_AUT0_DIRECT_L09
\   0023  FCFF              DEFW     -4
   16             int inc_var-l;
\   0025  DD36FC01          LD       (IX-4J.1
\   0029  DD36FD00          LD      (IX-3) .0
```

```
  17            char inc_char;
  18            inc_char-'b';
\   002D  DD36FE62          LD      (IX-2),98
  19          }
\   0031  C30000            JP      ?LEAVE_DIRECT_L09
\   0034            main:
  20
  21          void main(void)
  22          {
\   0034  CD0000            CALL    ?ENT_AUT0_DIRECT_L09
\   0037  FEFF              DEFW    -2
  23            int my_int-0:
\   0039  AF                XOR     A
\   003A  DD77FE            LO      (IX-2).A
\   0030  DD77FF            LD      (IX-l).A
  24            call_count-0;
\   0040  210000            LD      HL.O
\   0043  220000            LD      (call_count),HL
  25            my_char-con_char;
\   0046  3A00O0            LD      A,(con_char)
\   0049  320200            LD      (my_char).A
  26            set_local():
\   004C  CD2000            CALL    set_local
\   004F            ?0001:
  27            while (my_int<100)
\   004F  016480            LD      BC.32868
\   0052  DD6EFE            LD      L.UX-2)
\   0055  DD66FF            LD      H.(IX-l)
\   0058  3E80              LD      A,-128
\   005A  AC                XOR     H
\   005B  67                LD      H.A
\   005C  ED42              SBC     HL.BC
\   005E  300D              JR      NC,70000
\   0060            ?0002:
  28            {
  29             do_foreground_process();
\   0060  CD0000            CALL    do_foreground_process
  30             my_int++:
```

```
\   0063  DD34FE          INC     (IX-2)
\   0066  2003            JR      NZ.70003
\   0068  DD34FF          INC     (IX-1)
\   006B            70003:
  31          }
  32         }
\   006B  18E2            JR      70001
\   006D            70000:
\   006D  C30000          JP      ?LEAVE_DIRECT_L09
\   0000                  RSEG    CONST
\   0000            con_char:
\   0000  61              DEFB    'a'
\   0000                  RSEG    UDATAO
\   0000            call_count:
\   0002                  DEFS    2
\   0002            my_char:
\   0003                  DEFS    1
\   0003                  END
```

LINKING THE PROGRAM

To link the program, enter the command:

```
xlink tutorl -f lnk -rt -x -1 tutorl.map 0
```

The -f option specifies your XLINK command file 1 nk, and the -rt option allows the code to be debugged using C-SPY (the t indicates that C-SPY's terminal I/O routine will be used to display output).

The -x creates a map file and the -1 *fi lename* gives the name of the file.

The result of linking is a code file called a out. dOl and a map file called tutorl.map.

Examine the map file to see how the segment definitions and code were placed into their physical addresses. The most important information about segments is at the end where the address and range are given:

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

SEGMENT	START ADDRESS		END ADDRESS	TYPE	ORG	P/N	ALIGN
RC0DE	0100	-	017C	rel	stc	pos	0
CODE	017D	-	01F1	rel	fit	pos	0
CDATA0	Not in use			rel	fit	pos	0
ZVECT	Not in use			rel	fit	pos	0
CONST	01F2	-	01F2	rel	fit	pos	0
CSTR	Not in use			rel	fit	pos	0
CCSTR	Not in use			rel	fit	pos	0
INTVEC	Not in use			com	stc	pos	0
DATA0	Not in use			rel	stc	pos	0
IDATA0	Not in use			rel	fit	pos	0
UDATA0	C000	-	C002	rel	fit	pos	0
ECSTR	Not in use			rel	fit	pos	0
WCSTR	Not in use			rel	fit	pos	0
TEMP	Not in use			rel	fit	pos	0
CSTACK	C003	-	C202	rel	fit	pos	0
	0000	-	0002	aseg			

Notice that, although the link file specified the address for all segments, many of the segments were not used.

Several entry points were described that do not appear in the original C code. The entry for ?c_exi t is from the CSTARTUP module. The putchar entry is from the library file (since we have used the - rt flag, the C-SPY debug putchar code is used).

RUNNING THE PROGRAM

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

cs64180s aout 0

At the C-SPY prompt, enter the command:

STEP 0

Repeat this until the line reading do_f oreground_process ( ); is highlighted.

Now, to inspect the value of the variable cal 1_count, enter:

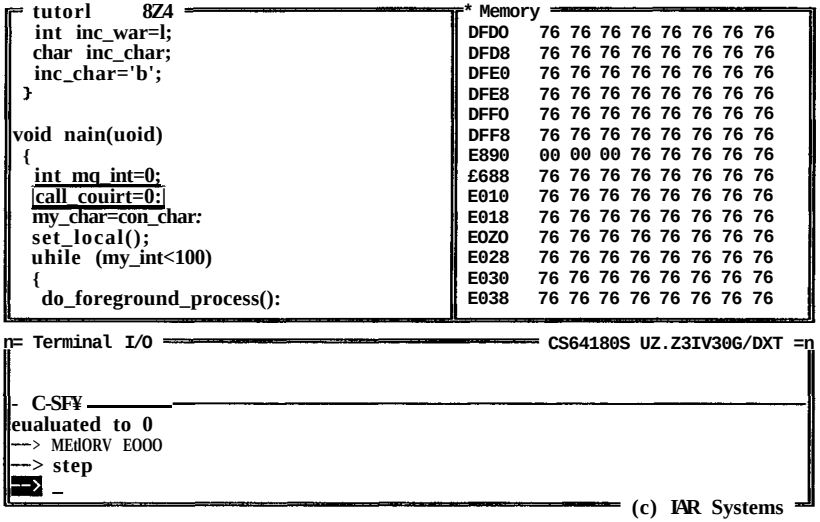
EXPR call_count 0

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

Now, enter the commands

MEMORY E000 0
STEP 0

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



This displays the current contents of memory from address E000 (where the variables are located). The next step executes the current line and moves to the next line in the loop. Now examine the variable again by entering:

```
EXPR  ca11_count  0
```

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

Enter:

```
LEVEL  0
STEP  0
```

The assembler code for the C program is displayed and the steps are by assembler lines, rather than by C statements.

Note that the address of the code is based on the link file specification.

Enter:

```
LEVEL  0
STEP  11  0
```

This returns to C level and steps through 11 instructions.

You can modify variables or memory contents while you are debugging. Enter:

```
EXPR  my_char='c'  0
STEP  11  0
```

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

QUITTING C-SPY

To quit C-SPY, enter the command:

```
QUIT  0
```

MODIFYING THE COMPILE AND LINK OPTIONS

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

- Use the - v 0 processor option instead of the - v 1 option for compiling. Use the link file 1 nkz80.xcl instead of 1 nk.xcl. Notice that the variable addresses have changed.

USING THE C EXIT ROUTINE

If you are testing the program in C-SPY, continually step through the program until the counter my_i nt eventually reaches 100 and the C program exits. Use the C-SPY command:

STEP 30 0

to repeatedly step through the program 30 lines at a time.

There is a entry point for exi t in the linked code. You can create your own exit processing, but normally you will wish to write the C code such that it remains in an infinite loop.

EXTENDING THE PROGRAM

We shall now extend the program to access the serial I/O channel built in to the 64180 microprocessor. The resultant program accepts input from serial port number 0 and outputs the character to a port. This serial program demonstrates the use of the #pragma directive and inclusion of supplied header files.

The following is a complete listing of the program. Enter it into a suitable text editor and save it as tutor2. c. Alternatively, a copy is provided in the c:\iar\iccz80 directory:

```
 /* enable use of extended keywords */
#pragma language=extended
 /* include definitions for 10 registers */
#indude <io64180.h>
```

```
 /* include definitions for intrinsics */
#include <intrz80.h>
 /* combines input function and mask */
#define receive_full (STAT0 & 128)
 /* output port decoded by external logic */
#define my_port  0xC0
/***********************************
*                                  *
*        Start of code             *
*                                  *
***********************************/
char my_char;
int call_count;

char read_char(void)
{
  /* Loop until bit 7 indicates receive data */
  while (!receive_full);
  my_char= RDR0;
  /* return receive register */
  return(my_char);
}

void do_foreground_process(void)
{
  call_count++;
  output8(my_port,my_char);
}

void main(void)
{
  /* Initialize comms channel */
  /* 2400 with 6.144 MHz clock */
  CNTLB0 - 0x2A;
  /* enable and data format */
  CNTLA0 = 0x65;

  /* now loop forever, taking input when ready */
```

```
  while (1)
  {
    if (read_char()) 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 1-82.

The second line of code is:

```
#include <io64180.h>              /* include definitions for
                                   10 registers */
```

The file i O64180. h includes definitions for all I/O registers for the 64180 microprocessor.

The full list of C source header files for processor I/O is given in *C include files*, page 1-12.

The line below set up and test the serial input:

```
#define receive_full (STATO & 128)
 /* output port decoded by external logic */
. . .
  while (!receive_full);
  my_char= RDRO;
```

Bit 7 of STATO, one of the I/O registers defined in the included header file, indicated a character has been received.

```
 /* Initialize comms channel */
 /* 2400 with 6.144 MHz clock */
 CNTLBO = 0x2A;
 /* enable and data format */
 CNTLAO = 0x65;
```

The bits set in CNILBO set the baud rate to be 2400.

COMPILING AND LINKING THE SERIAL PROGRAM

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

```
iccz80 tutor2 -ml -r -L -q -vl 0
xlink tutor2 -f lnk -rt 0
```

RUNNING THE SERIAL PROGRAM

As before, to run the program, enter:

```
cs64180 aout 0
```

In C-SPY enter the command:

```
STEP 0
LEVEL 0
```

and step until C-SPY fails to give a prompt. The program is now waiting in a while loop for an external input.

Stop execution and simulate an input by entering:

```
@C
sfr 4=80 0
sfr 8=63 0
window register ON 0
STEP 0
```

On a real target with the serial input port number connected to a transmitter, a received character would set the 'byte received' flag in bit 7 of the serial control register and be transferred to the data register. The steps above force the ready bit to 1.

Now enter:

```
STEP 0
ISTEP 0
```

several times and watch the program sequence through its code and place the character it reads from the serial register to the output port.

Use care when designing your test programs to avoid tight loops since they can be very difficult to test. Place dummy function calls or assignments inside the loops to simplify debugging.

EXITING FROM C-SPY

Quit C-SPY by entering:

```
QUIT 0
```

ADDING AN INTERRUPT HANDLER

We shall now modify the first tutorial 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 serial interrupt.

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

```
 /* enable use of extended keywords */
#pragma language=extended
 /* include definitions for 10 registers */
^include <io64180.h>
 /* include definitions for intrinsics */
#include <intrz80.h>
 /* combines input function and mask */
#define receive_full (STATO & 128)
 /* output port decoded by external logic */
#define my_port  OxCO
/**•**********************************
*                                    *
*         Start of code              *
*                                    *
************************************/

char my_char;
```

```
int call_count;

interrupt [0x2e] void receive_char( void)
{
  /* interrupt indicates received data */
  my_char= RDRO;
  /* output receive register */
  output8(my_port,my_char);
}

void do_foreground_process(void)
{
  call_count++;

}

void main(void)
{
  /* Initialize comms channel */
  /* 2400 with 6.144 MHz clock */
  CNTLBO = 0x2A;
  /* enable and data format */
  CNTLAO = 0x65 ;
  /* enable interrupts on received char */
  STAT0|-1;
  /* enable interrupts through C function */
  /* INTVEC was BF in link file */
  /* Set up IL register to point at table */
  output8(IL. 0x20);
  load_I_register(0x3f);
  enable_interrupt();
  /* now loop forever, taking input automatically */
  while (1)
  {
    do_foreground_process();
  }
}
```

The interrupt include files must be present to define the registers used:

The following lines initialize the interrupt registers and enable interrupts:

```
/* enable interrupts on received char */
STAT0|=1;
/* enable interrupts through C function */
enable_interrupt();
```

The interrupt function itself is defined by the following lines:

```
interrupt [0x2e] void receive_char( void)
{
  /* interrupt indicates received data */
  my_char= RDR0;
  /* output receive register */
  output8(my_port,my_char);
}
```

This function is called whenever there is a received character. The response of this program is to output a character, making it easy to identify the event. If you were working with a real target that had a diagnostic output channel, you would probably output some diagnostic information and then abort.

The `interrupt` keyword is described in *Extended keyword reference*, page 1-85.

COMPILING AND LINKING THE PROGRAM

Compile and link the program as before:

```
iccz80 tutor3 -r -L -q -vl 0
xlink tutor3 -f lnk -r 0
```

RUNNING THE INTERRUPT PROGRAM

As before, to run the program, enter:

```
cs64180s  aout  0
```

followed by:

```
STEP  0
LEVEL  0
```

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

EXITING FROM C-SPY

Quit C-SPY by entering:

```
QUIT  0
```

USING BANKED MEMORY

If the target system has ROM bank switching, the C program can make use of this for code storage. The example below shows how the non_banked keyword can be used to select functions that should remain in the main bank.

```
 /* enable use of extended keywords */
#pragma language=extended
 /* include definitions for 10 registers */
#include <io64180.h>
 /* include definitions for intrinsics */
#include <intrz80.h>
/***********************************
*                                  *
*         Start of code            *
*                                  *
••it*********************************/
extern char my_char;
```

```
extern non_banked void do_foreground_process(void);

void init_comms(void)
{
  /* Initialize comms channel */
  /* 2400 with 6.144 MHz clock */
  CNTLB0 = 0x2A;
  /* enable and data format */
  CNTLA0 = 0x65 ;
  /* enable interrupts on received char */
  STAT0|=8;
  /* enable interrupts through C function */
  /* INTVEC was 3F in link file */
  /* Set up IL register to point at table */
  output8(IL. 32);
  load_I_register(0x3f);
  enable_interrupt();
}

void main(void)
{
  init_comms();
  while (1)
  {
    /* now loop forever, taking input automatically */
    do_foreground_process();
  }
}
```

This program is provided in the sample tutorials astutor4a.c, Compile the program using the -mb memory model.

```
iccz80 tutor4a -mb -r -q -L -vl
```

Examine the extract from the list file below and note how the calling methods for a banked and an unbanked function differ.

```
 30          void main(void)
 31           {
\  0016             main:
 32            init_comms();
\  0016  3E00              LD      A,BYTE3 init_comms
\  0018  210000            LD      HL,init_comms
\  001B  CD0000            CALL    ?BANK_CALL_DIRECT_L08
\  001E            ?0001:
 33            while (1)
 34             {
 35               /* now loop forever, taking input */
 36               do_foreground_process();
\  001E  CD0000            CALL    do_foreground_process
 37             }
 38           }
\  0021  18FB              JR      70001
```

The non-banked function is called directly, but ?BANK_CALL_DI RECT_L08 stores the bank number of the address pointer first, then calls a routine to switch banks and jump to the address in AH L.

The banked memory model link files (1 n k64b or 1 n kz80b) must be used with a program compiled as banked. The banked function address is based on a three-byte pointer where bytes 0-1 provide the offset and byte 2 is the bank number to supply to the mapping circuit.

The 1nk64b.xc1 file provides a sample of how to define the external banks. The statement:

-b(CODE)CODE=4C4000,4000,40000

indicates that banks start at logical address 0x4000 with a size of 0x4000. The third is the bank size multiplied by 0x10.

The non-banked functions used by tutor4a. c are in tutor4b. c and tutor4c. c and are listed below:

```
 /* tutor4b.c */
 /* enable use of extended keywords */
#pragma language=extended
 /* include definitions for 10 registers */
```

```
#include <io64180.h>
 /* include definitions for intrinsics */
#include <intrz80.h>
 /* combines input function and mask */
#define receive_full (STATO & 128)
 /* output port decoded by external logic */
#define my_port  OxCO
/***********************************
*                                  *
*          Start of code           *
*                                  *
***********************************/
char my_char;

#pragma function=non_banked
interrupt [0x2e] void receive_char( void)
{
  /* interrupt indicates received data */
  my_char= RDRO;
  /* output receive register */
  output8(my_port,my_cha   r);
}
```

Compile tutor4b in the same way as the main module:

```
iccz80 tutor4b -mb -r -x -L -vl
```

```
/* tutor4c.c */
 /* enable use of extended keywords */
#pragma language=extended
 /* include definitions for 10 registers */
#include <io64180.h>
 /* include definitions for intrinsics */
#include <intrz80.h>
 /* combines input function and mask */
#define receive_full (STATO & 128)
 /* output port decoded by external logic */
#define my_port OxCO
```

```
/***********************************
*                                  *
*          Start of code           *
*                                  *
***********************************/
char my_char;

#pragma function=non_banked
interrupt [0x2e] void receive_char( void)
{
  /* interrupt indicates received data */
  my_char= RDRO;
  /* output receive register */
  output8(my_port,my_char);
}
```

Compile tutor4c in the same way as the other modules:

```
iccz80 tutor4c -mb -r -x -L -vl
```

Link all three source files by entering:

```
xlink tutor4a tutor4b tutor4c -f lnk -rt -x -1 tutor4.map
```

Extracts of the map file for the combined program is shown below:

```
DEFINED ABSOLUTE ENTRIES
PROGRAM MODULE, NAME : ?ABS_ENTRY_MOD
        ABSOLUTE ENTRIES        ADDRESS          REF BY MODULE
        ----------------        -------          -------------
        CBR                      0008            CSTARTUP
        CBAR                     0084            CSTARTUP

  ************************************************************************
```

The startup routine sets up the address banking variables.

```
FILE NAME : tutor4a.r01
PROGRAM MODULE. NAME : tutor4a
```

```
  SEGMENTS IN THE MODULE
  ----------------------

CODE
  Banked segment, address : 004C:4000 - 004C:4023
           ENTRIES                ADDRESS          REF BY MODULE
           im't_comms              004C:4000       Not referred to
           main                    004C:4016       CSTARTUP
           LOCALS                 ADDRESS
           ?0001                   004C:401E

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


  FILE NAME : tutor4b.r01
  PROGRAM MODULE. NAME : tutor4b

  SEGMENTS IN THE MODULE
  ----------------------

RCODE
  Relative segment, address : 0100 - 0107
           ENTRIES                ADDRESS          REF BY MODULE
           do_foreground_process   0100            tutor4a
    --------------------------------------------------------------------------------
UDATAO
  Relative segment, address : 8000 - 8001
           ENTRIES                ADDRESS          REF BY MODULE
           call_count              8000            Not referred to

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


  FILE NAME : tutor4c.r01
  PROGRAM MODULE, NAME : tutor4c

  SEGMENTS IN THE MODULE
  ----------------------

RCODE
  Relative segment, address : 0108 - 0119
           ENTRIES                ADDRESS          REF BY MODULE
           receive_char            0108            Not referred to
```

```
   ------------------------------------------------------------------------
INTVEC
  Common segment, address : 3F00 - 3F0C
   ------------------------------------------------------------------------
UDATAO
  Relative segment, address : 8002 - 8002
          ENTRIES                ADDRESS          REF BY MODULE
          my_char                 8002            Not referred to

   ************************************************************************
```

The entries show the module addresses and which modules call this module.

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

SEGMENT	START ADDRESS		END ADDRESS	TYPE	ORG	P/N	ALIGN
DATAO	Not in use			rel	stc	pos	0
IDATAO	Not in use			rel	fit	pos	0
UDATAO	8000	-	8002	rel	fit	pos	0
ECSTR	Not in use			rel	fit	pos	0
WCSTR	Not in use			rel	fit	pos	0
TEMP	Not in use			rel	fit	pos	0
CSTACK	8003	-	8202	rel	fit	pos	0
RCODE	0100	-	019E	rel	stc	pos	0
CDATAO	Not in use			rel	fit	pos	0
ZVECT	Not in use			rel	fit	pos	0
CONST	Not in use			rel	fit	pos	0
CSTR	Not in use			rel	fit	pos	0
CCSTR	Not in use			rel	fit	pos	0
INTVEC	3F00	-	3F0C	com	stc	pos	0
CODE	004C:4000	-	004C:4023	bnk	fit	pos	0
	0000	-	0002	aseg			

The segments used and their addresses are listed at the end of the map.

If you use banked memory with the z80, you must provide decoding circuitry for the ROM banks. The high order address lines (A 12 to A15) must be decoded to select which area of memory is being accessed, and an additional decoder (driven by an I/O port for example) must provide the bank number.

For more information about using banked memory see *Banked memory,* page 1-63, and *Configuration,* page 1-59.

You may want to split your program into several C or assembler modules to simplify maintenance. See *Configuration,* page 1-59, for more information on using multiple modules.

MODIFYING CSTARTUP

A standard C setup module is provided with the compiler, but you may wish to create your own.

For example, the startup code could change the I/O register or page zero memory addresses. It might be desirable to include initialization routines in CSTARTUP rather than in the main code if your implementations always use a standard environment.

When the C program is compiled, it is executed after an initialization routine, but before an exit routine. Normally the exit routine is never called as the code is used in a dedicated controller which loops continuously. If you want to include special startup or shutdown code, edit the CSTARTUP file, using the CSTARTUP. SOI assembly source module as a starting point for modification. The code is commented to indicate what action is taking place during the execution:

- Stack pointer initialized.
- Data memory initialized.
- C ma i n called.
- Jump to exit routine.

Add code at the appropriate point to initialize your hardware and assemble the modified source file. After successful assembly, use XLIB to place the new module in the library file for the processor and memory type (for example, cl z80. sOl for the Z80).

The code below shows several important sections of the startup file.

```
#define  proc64180  (__TID_&0x010-0x010)

          NAME    CSTARTUP

          EXTERN          main                      ; where to begin
execution
          EXTERN          ?C_EXIT                   ; where to go when
program is done

#ifdef banking
#if proc64180

CBAR_addr EQU     3AH             ; define I/O ports to MMU registers
CBR_addr  EQU     38H             : (See also defines in debug.sOl and
108.sOl)

          EXTERN   CBAR
          EXTERN   CBR
#endif

          EXTERN          ?BANK_CAIL_DIRECT_L08
#endif

;---------------------------------------------------------------;
;  CSTACK - The C stack segment                                 ;
;                                                               ;
;  Please, see in the link file lnk*.xcl how to increment       ;
;  the stack size without having to reassemble cstartup.sOl !   ;
;---------------------------------------------------------------;

          RSEG    CSTACK
          DEFS    0                         ; a bare minimum !
```

```
;-------------------------------------------------------------;
;  Forward declarations of segment used during initialization  ;
;-------------------------------------------------------------;
          RSEG   UDATA0
          RSEG   IDATA0
          RSEG   ECSTR
          RSEG   TEMP
          RSEG   DATA0
          RSEG   WCSTR

          RSEG   CDATA0
          RSEG   ZVECT
          RSEG   CCSTR
          RSEG   CONST
          RSEG   CSTR

          ASEG
          ORG    0

          JP     init_C

;-------------------------------------------------------------;
;  RCODE - where the execution actually begins                ;
;-------------------------------------------------------------;
          RSEG   RCODE
init_C
          LD     SP,.SFE.(CSTACK-1)       ; from high to low address

;-------------------------------------------------------------;
; If hardware must be initiated from assembly or if interrupts ;
; should be on when reaching main, this is the place to insert ;
; such code.                                                   ;
;-------------------------------------------------------------;

#ifdef banking
#if proc64180
```

```
;-------------------------------------------------------------;
; Setting of MMU registers - see chapter "Linking" of manual.  ;
;-------------------------------------------------------------;

          LD      A.CBAR                ; set CBAR value
          OUTO    (CBAR_addr).A

          LD      A.CBR                 ; set CBR value
          OUTO    (CBR_addr).A

#endif
#endif

;-------------------------------------------------------------;
: If it is not a requirement that static/global data is set   ;
; to zero or to some explicit value at startup, the following  ;
; line referring to seg_init can be deleted, or commented.     :
;-------------------------------------------------------------;

          CALL    seg_init

#ifdef banking

          LD      HL.LWRD(main)         ; banked call to mainO
          LD      A,BYTE3(main)
          CALL    ?BANK_CALL_DIRECT_L08
#else
          CALL    main                  ; non-banked call to mainO
#endif

;-------------------------------------------------------------;
; Now when we are ready with our C program we must perform a   ;
; system-dependent action.  In this case we just stop.         ;
;-------------------------------------------------------------;
; DO NOT CHANGE THE NEXT LINE OF CSTARTUP IF YOU WANT TO RUN   ;
: YOUR SOFTWARE WITH THE HELP OF THE C-SPY HLL DEBUGGER.      ;
;-------------------------------------------------------------;
```

```
          JP      ?C_EXIT
. . .

?C_EXIT
exit      EQU     ?C_EXIT

;--------------------------------------------------------------;
; The next line can be replaced by user defined code.         ;
;--------------------------------------------------------------;
          JR      $                         ; loop forever

          END
```

The stack size can be set in CSTARTUP, but it is simpler to use the -Z(DATA)CSTACK+stack_size declaration in the linker filer. (Using the linker file avoids having to re-assemble CSTARTUP).

The initialization of an internal sfr register or an external output divide could be done after the section starting "If hardware must be initiated". For example, add the assembly code below to have the 64180 output to address I/O 101:

```
LD      A,101
OUT0    (192),A
```

If you have any other code you want executed before main starts, insert that here as well.

If you want to insert assembler shutdown code, insert that after ?C_EXIT.

ADDITIONAL EXAMPLES

The examples below demonstrate more advanced C programming techniques. There are also examples in other sections of the manual, particularly in *Data representation,* page 1-77, and *Assembly language interface,* page 1-117.

USING VARIABLES

This example shows a variety of ways of declaring variables. Compile as before and examine the list file. This example is available as exampl 1. c.

```
/* More uses of variables */
int staticint;
char staticchar;
const char conschar='a';
const int consint=3;
char * cstring="constant";
const char constring[]="rom";
no_init int noinitv;
unsigned char carr[0xl00];
struct
{
  short a:2;
  short :1;  /* gap in bitfield */
  short b:l;
} bf;

unsigned char * intram_0 = (unsigned char *) 0x100;
unsigned char * intram_l = (unsigned char *) 0x110;
char m_ram[]={ "Place your message here" };
const char m_rom[]={ "Fixed text" };

void main(void)
{
  char localint-1;
  char localchar='c';
  bf.a-1; /* set two bits of bf to 01 */
  * intram_0 = 0x80;
  strcpy(m_ram,m_roiTi);
}
```

USING MEMORY-MAPPED OUTPUT

This example shows how to use a pointer for memory-mapped output. Compile as before and examine the list file. This example is available as examp!2.c.

```
#include <stdio.h>
int putcharCint outchar)
{
  unsigned char *LCD_I0;
  LCD_I0= (unsigned char *) 0x8000;
  * LCD_I0=outchar;
  return(outchar);
}
```

USING POINTERS

This example shows more ways of using pointers. Compile as before and examine the list file. This example is available as exampl 3. c.

```
#pragma language=extended
char mychar;
/* global variable to hold char data from pointers */

char * gp;
char * pcd;
/* global pointers to character */

#define my_data ((char *) OxDOOO)

char text[10];

void main(void)
{
  char *lp;   /* local pointer to char */
  pcd="abcd"; /* pointer to string in ROM */
  pcd=(char *) OxDEEE;
  mychar= * my_data;
  gp=(char *) OxDEED;
  /* constant pointers to uninitialized char */
```

```
  text[0]= * pcd;
  putchar(* (text+1));
  putchar(* gp);
  putchar('a');
  pcd= gp;
  mychar=* pcd;
  pcd=text;
  mychar=* pcd;
  lp=my_data;
  * pcd= mychar;
  * pcd='a';
  * gp= 'a';
}
```

USING RECURSIVE FUNCTIONS

This example shows a recursive function. Compile as before and examine the list file. Note that the calling method is the same as for any function since ? ENT_PARM_D I RECT_L09 creates a new variable area on the stack for each call. This example is available as exampl 4. c.

```
void recursive(int value)
{
  int my_int;
  my_int=l;
  va1ue-=my_int;
  if (value>10) recursive(value);
}
void main(void)
{
  recursive(50);
}
```

USING PRAGMAS

The example below shows how to use pragmas to specify function and memory characteristics. This example is available asexamp!5.c.

```
/* using pragma definitions instead of modifiers */
#include <stdio.h>
#pragma language=extended
int ccount;
#pragma memory=dataseg(mydseg)
/* you must define mydseg in the linker file */
int loopc;
#pragma memory=default

#pragma function=monitor
void looplO(void)
{
  for (loopc=l;loopc<10;loopc++);
}
#pragma function=default

#pragma function=non_banked
void printcnt(int cent)
{
  printfC'the count is now %d",ccnt);
}
void main(void)
{
  int my_int=0;
  int my_int2=0;
  ccount=l;

  while (my_int<100)
  {
    looplOO;
    my_int++;
    printcnt(my_int);
  }
}
```

CONFIGURATION

INTRODUCTION

Systems based on the Z80 microprocessor family can vary considerably in their use of internal and external ROM and RAM, and in their stack requirements. They also differ in their need for reentrancy, large libraries, or time-critical functions. This chapter describes how to configure the IAR C Compiler for a given application.

The memory model and link options specify:

- The ROM areas: used for non-bankable functions, bankable functions, constants, and initial values.
- The RAM areas: used for directly addressable internal memory, indirectly addressable internal memory, external memory, and external non-volatile memory.

The compiler and linker identify these different types of memory by giving them different segment names , such as RCODE for code in ROM and IDATA for data in internal RAM.

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	Compiler keyword, Linker command file.
Optimization	Compiler option.
putchar and getchar functions	Run-time library module.
pri ntf/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 features of the system.

There is an alternative run-time library for each combination of processor group and memory model.

	Memory model	
Processor	*Large*	*Banked*
Z80	clz80.r01	Clz80b.r01
HD64180orZ8018X	Cl64180.r01	cl64180b.r01

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

LINKER COMMAND FILE

To create a linker command file for a particular project the user first copies the supplied template c:\iar\iccz80\1nk.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 IAR C Compiler supports four memory models. These offer a choice of default placement for local and global variables within the ROM (CODE) and RAM (DATA) memory.

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*
-m1	Large memory model
-mb	Banked memory model

For example, to compile myprog in the banked memory model, use the command:

```
iccz80 myprog -mb
```

If you include none of the memory model options, the compiler uses the large memory model.

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

	Large	*Banked*
Z80	cU80.r01	clz80b.r01
Z80180/64180	Cl64.r01	Cl64b.r01

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

```
xlink myprog -f lnk cl80b
```

The -f option specifies a command filename which holds the assignments for the segment areas; see the *IAR Z80/64180 Assembler* guide for details.

LINKER COMMAND FILE

Four standard linker command files have been supplied with the compiler. To create a linker command file for a particular project, copy the supplied template c:\iar\iccz80*.xcl and modify the file to specify the details of the target system's memory map. Instructions for modifications are included within the files.

The supplied linker files have the following characteristics:

File	*lnkz80*	*lnkz80b*	*lnk64*	*lnkz80b*
Processor library	clz80	clz80b	cl64180	cl 64180b
ROM area	100-BEFF	100-3EFF	100-BEFF	100-3EFF
Banked area	-	4000-7FFF	–	4000-7FFF
RAM area	C000-FFFF	8000-FFFF	C000-FFFF	8000-FFFF
INTVEC	BFOO	3F00	BFOO	3F00
Compile options	-ml -vO	-mb -vO	-ml -vl	-mb -vl

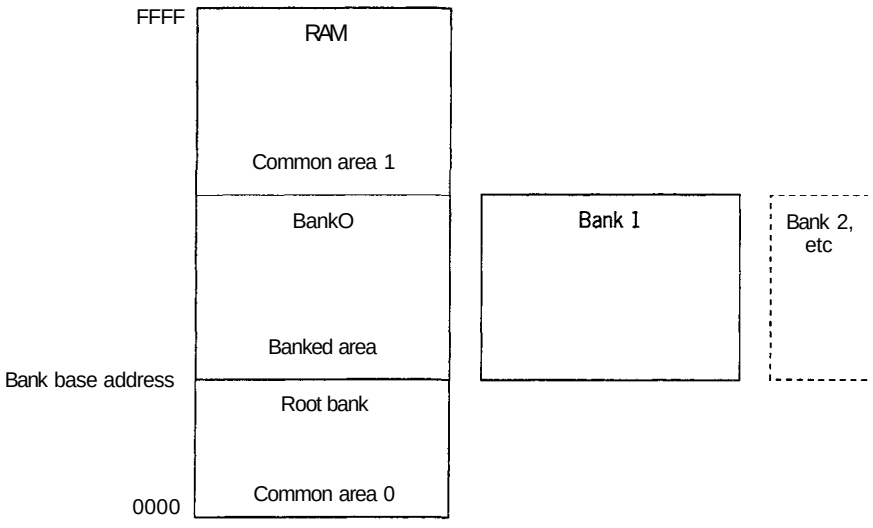
MEMORY LOCATION

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 the guide *IAR Assembler, Linker, &. Librarian for the Z80/64180.*

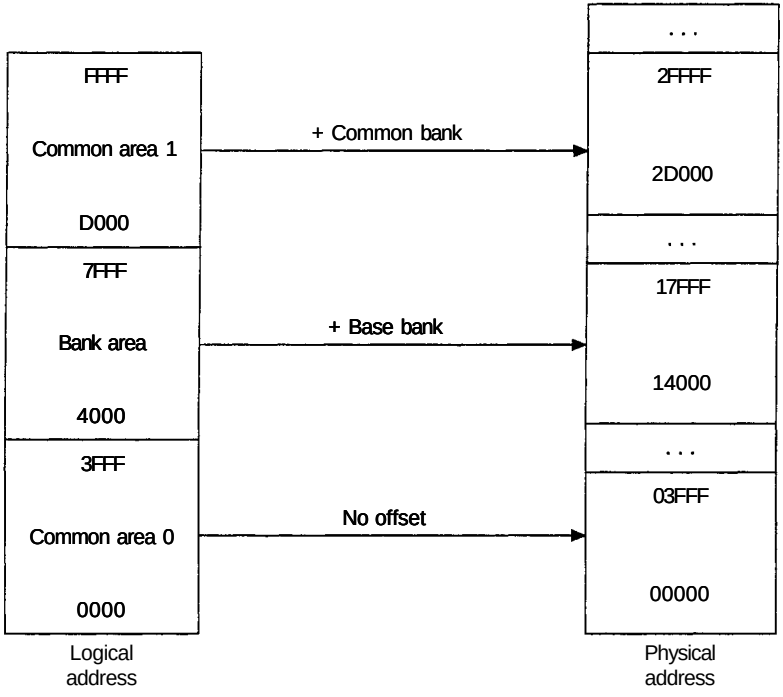
BANKED MEMORY

The microprocessor can use a banked CODE area as shown below:



There can be up to 256 banks of memory in addition to the root bank.

The logical addresses are mapped into a 512 Kbyte physical address space by the base registers:



A banked memory model link file (used with linker files 1 nk64b or 1 n kz80b) must be used with a program compiled as banked. The banked function address is based on a three-byte pointer where bytes 0-1 provide the offset and byte 2 is the bank number to supply to the mapping circuit.

The 1 nk64b. xcl file provides a sample of how to define the external banks:

```
-b(CODE)C0DE=4C4000,4000,40000
-DCBAR=84
-DCBR=8
```

- The -b indicates that this CODE segment information is for banked memory.

- The first two characters of the first parameter (4C) indicates the starting bank address is 4C0000.
- The second half of the first parameter (4000) gives the starting address of the first bank as 0x4000.
- The second parameter indicates that the bank window size is 0x4000.
- The third parameter is the bank window size multiplied by 0x10.
- The first character in the CBAR specification is the hex character identifying the 4 Kbyte boundary of the start of common area 1.
- The second character in the CBAR specification is the hex character identifying the 4 Kbyte boundary of the start of the banked area. (The first character must be greater than the second character. The common area is always higher in the address space than the banked area.)
- The character in the CBR specification is the hex character identifying the start of the data area as 8000.

CHANGING THE BANKED MEMORY SPECIFICATION

If your hardware does not use the values specified in the banked link files, you will have to calculate the correct values for your system.

CBAR is the two hex characters representing the physical page boundaries of the memory system. The first character is the start of the bank area and the second is the start of common area 1.

BBR is calculated by XLINK and does not have to be specified. It is the value to be placed into the current bank register. This number is multiplied by 0x1000 and added to the current address when the logical address is inside the bank area.

To calculate CBR, take the physical address of the data area, subtract the logical address, and divide by 0x1000. This value must be specified in the link file.

The -b(CODE) specification is based on the memory boundaries and the bank window size.

-b(C0DE)C0DE=cb5SSS, *wwww, wwwwO*

cb is the value used for the BBR register

ssss is the logical start address

wwww is the bank window size

wwwwO is the bank window size multiplied by Oxl 0.

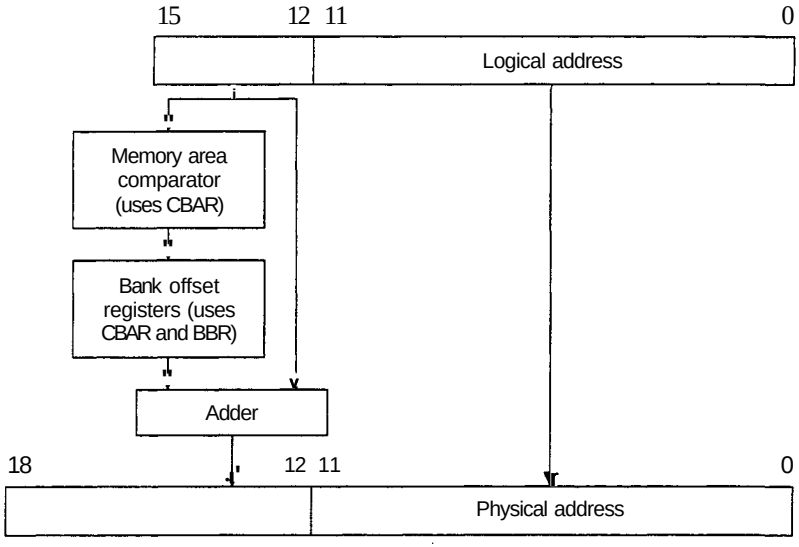
The physical address of code in memory is then c6*0xl000+ssss.

The utility bnksetup. exe will calculate the segment definitions and the values of CBAR and CBR from the physical and logical addresses you supply to it. Use this as a starting point for creating your linker command file.

There are additional points to remember when designing a system using banked memory:

- No single module can be bigger than the bank size.
- The compiler will select the fastest calling method for a function. A function can be forced into the non-banked portion of memory if its execution must be done as quickly as possible. An example of this is the 1 o op 10 function in the tutorial; see *Using banked memory*, page 1-42.
- Interrupt functions cannot be banked, but they can call banked functions.
- You can use as many - b lines in your link file as you require. If you use one line per segment, each line with its own set of parameters, you can assign code to a particular bank.

The 64180 has bank switching circuitry included.



If you use banked memory for the Z80, you must provide decoding circuitry for the ROM banks. The high order address lines (A 12 to A15) must be decoded to select which area of memory is being accessed, and an additional decoder (driven by an I/O port for example) must provide the bank number.

MULTI-MODULE LINKING

You may want to split your program into several C or assembler modules to simplify maintenance. The entry to your modules should be declared in the C main block as shown below:

```
extern int mymodl (int myintl, int myint2);
int total;
void main(void)
{
  total=mymodl (3, 4);
}
```

A separate file will contain the code for mymod 1, for example as:

```
int mymodKint paral, int para2)
{
  return (paral+=para2)
}
```

You can also use the module to create a shell for an assembler routine.

Compile the two modules in the regular way with the same memory model.

Place the objects resulting from the compilations into a library by using XLIB, as follows:

```
XLIB
* def-cpu z80
* fetch-mod main mylib
* fetch-mod mymodl mylib
* list-mod mylib mylib.1st
* exit
```

Modify a copy of the linker file to include access to the library and add your output file specifications. Name the new linker file mymod. xcl.

```
-! link the library module which contains our object modules
-j
my! ib
-! specify the output file name -!
-o mulmod.aOl
```

The linker file can now use the modules in the library and produce an executable program. Use the linker with the new control file:

```
xlink -f mymod
```

NON-VOLATILE RAM

The compiler supports the declaration of variables that are to reside in non-volatile RAM through the no_i ni t type modifier and the memory #pragma. The compiler places such variables in the separate segment no_i n i t, 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_i ni t 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 the guide *IAR Assembler, Linker, & Librarian for the Z80/64180.*

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.

OPTIMIZATION

USING THE ALTERNATIVE REGISTER SET

The Z80 has an alternative register set which can be reached by using the EXX and EX AF, AF' instructions. The compiler has two alternative uses for these registers: fast interrupts and normal code.

If the -ua command line option is given, the alternative register set is allocated for normal code. This means that:

- You cannot have interrupts using ALTERNATE_SET. If the compiler detects that you have enabled this switch and declared an interrupt using ALTERNATE_SET you will get an error message.
- If you mix EXX usage in different modules, the linker will not detect it. Ensure that your code modules do not specify incompatible alternate register usage.
- The code generator will use a slightly faster bank call mechanism that uses the AF' register.
- The code generator may allocate HL', DE', and BC' for register variables. This will result in more compact and faster code.
- Interrupts that call functions will slow down as the code generator must emit code to preserve the alternative register on the stack.

USING RST VECTORS

The RST vectors can be used by the compiler for frequent library calls to reduce the code size. To select this option give the -ur command line option.

With this option in use, the RST vectors are assigned as follows:

Vector	*Description*
00	Not used (reset).
08	Function enter with parameters.
10	Function enter with autos and possible parameters.
18	Bank leave.
20	Non banked leave.
28	Bank call.
30	Reserved for emulator/monitor breakpoints.
38	Reserved for IM1 interrupts.

USING UNDOCUMENTED INSTRUCTIONS

The -uu command line option allows the compiler to take advantage of undocumented Z80 instructions. A full list of these instructions is given in the file az80ext.s01.

The result on the code is that IXL, IXH, IYL, and IYH maybe allocated for character register variables. This will reduce the code size and improve speed.

Note that these are undocumented instructions and may not work on all manufacturers' versions of the Z80, since manufacturers do not guarantee the operation of undocumented instructions. It is important to check the operation of any code using undocumented instructions on the particular version of the Z80 to be used.

CHARACTER INPUT AND OUTPUT

PUTCHAR AND GETCHAR

The functions putcha r and getcha r 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\iccz80\putchar.cand c:\iar\iccz80\getchar.cThe 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 large memory model, compile putchar.c for the large memory model with the command:

```
iccz80 putchar -ml -z9
```

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

- Add the new p u t c h a r module to the appropriate run-time library module, replacing the original. For example, to add the new putcha r module to the standard large memory model library, use the command:

```
xlib
def-cpu z80
rep-mod putchar dz80
exit
```

The library module cl z80 will now have the modified putcha r instead of the original.

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

```
-A putchar
c!z80
```

This causes your version of putchar.rOl to load instead of the one in the cl z80 library; see the *IAR Z80 Assembler* guide.

The same procedure is also used for getchar.

Note that putchar serves as the low-level part of the pri ntf function.

PRINTF AND SPRINTF

The pri ntf and spri ntf functions use a common formatter called _formatted_wri te. The ANSI standard version of _formatted_wri te 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 _f ormatted_wri te, 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!
```

_medi um_wri te is considerably smaller than _f ormatted_wri te.

_small_write
As for _medi um_wri te, except that it supports only the *%%*, %d, %o, *%c*, *%s* and %x specifiers for i nt objects, and does not support f i el d wi dth and precision arguments. The size of _smal l_writeis 10-15% of the size of _formatted_write.

The default version is _small_wri te.

SELECTING THE WRITE FORMATTER VERSION

The selection of a write formatter is made in the linker control file. The default selection, _smal1_write, is made by the line:

```
-e_small_write=_formatted_write
```

To select the full ANSI version, remove this line.

To select _medi um_wri te, replace this line with:

```
-e_medium_write=_formatted_write
```

REDUCED PRINTF

For many applications spri ntf is not required, and even pri ntf with _sma 11 _wri te 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 pri ntf function (without spri ntf) is supplied in source form in the file i ntwri. 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 put char, above.

SCANF AND SSCANF

In a similar way to the printf and sprintf functions, scanf and sscanf use a common formatter called _f ormatted_read. The ANSI standard version of _f ormatted_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 _f ormatted_read, except that no floating-point numbers are supported. _medi um_read is considerably smaller than _f ormatted_read.

The default version is _medi um_read.

SELECTING READ FORMATTER VERSION

The selection of a read formatter is made in the linker control file. The default selection, _medi um_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 mallocorcalloc are used in the program, the C compiler creates a heap of memory from which their allocations are made.

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.

cstartupis also responsible for receiving and retaining control if the user program exits, whether through exi t or abort.

The user may wish to modify est a rtup, for example to initialize special hardware before entry to mai n, or to remove unwanted initialization of variables.

The overall procedure for modifying cstartupisas follows:

- Make the required modifications to the assembler source of cstartup, supplied by default in the file c: \ i a r\i ccz80\csta rtup. sOl, and save it under the same name.
- Assemble the modified cstartup using the appropriate memory model. For example, if the user program uses the large memory model, reassemble esta rtup for the large memory model with the command:

```
az80 cstartup.sOl
```

This will create a replacement object module file named cstartup.rO1.

- 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 large memory model library, use the command:

```
xlib
def-cpu z80
rep-mod cstartup clz80
exit
```

The library module cl z80 will now have the modified cstartup instead of the original.

Note that XLINK allows you to test the modified cstartup before installing it in the library by using the -C option. Add the following lines to the .xcl link file:

```
myestart
-C clz80
```

The CSTARTUP in mycstart will load instead of the CSTARTUP program module located in the cl z80 library. See the *IAR Z80 Assembler* guide for details.

DATA REPRESENTATION

DATA TYPES

The ICCZ80/64180 C Compiler supports all ANSI C basic elements. Variables are stored with the least significant part located at low memory address.

Variables are always tightly packed in memory and in structures, as the processor architecture does not require any particular alignment.

Data type	*Bytes*	*Range*	*Notes*
char (by default)	1	0 to 255	Equivalent tounsigned char
char (using -c option)	1	-128 to 127	Equivalent to si gned 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, data	2	0 to 65535	

Data type	*Bytes*	*Range*	*Notes*
pointer, code, using large memory model	2	0 to 65535	
pointer, code, using banked memory model	3	0 to 16777216	
float, double, long double	4	±1.18E-38to±3.39E + 38	

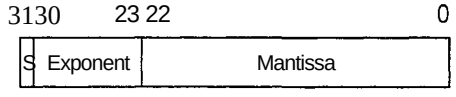
ENUMTYPE

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

FLOATING POINT

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

The memory layout of floating-point numbers is:



The value of the number is:

(-l)S * $_2$(Exponent-127) * i.Mantissa

Zero is represented by 4 bytes of zeros.

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

POINTERS

Data pointers are always 2 bytes.

Code pointers are as follows:

Keyword	*Storage in bytes*	*Memory model*
non_banked	2	Large
banked	3	Banked

DATA BANKING

Banked data is not supported directly as a data type, but it is possible to copy data from code banks. The intrinsic address_24_of returns a long pointer to the code bank holding the data; see *Intrinsic function reference*, page 1-109, for a detailed description of this intrinsic function.

EFFICIENT CODING

In order to get efficient code the following recommendations should be followed when possible:

- Use the smallest possible data types: the Z80/64180 is most efficient with byte types and rather inefficient with 4-byte types.
- Use unsigned data types whenever possible. The Z80/64180 generally performs unsigned operations more efficiently than the signed counterparts. This applies particularly to type conversions, comparison, array indexing and some arithmetic operations, such as » and /.
- Use the optimization options if possible. See *Optimization*, page 1-69, for details.

LANGUAGE EXTENSIONS

The IAR C Compiler provides a number of powerful extensions that support specific features of the Z80/64180 microprocessors.

The Z80/64180 extensions are provided in three ways:

- Extended keywords. By default, the compiler conforms to the ANSI specifications and Z80/64180 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. #pragma directives are instructions to target-specific compilers which provide additional information on how memory should be allocated, if extended keywords are recognized, and if warning messages are output.
- 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 is determined by the memory model chosen. In banked mode, the program may achieve additional efficiency for special cases by overriding the default by using the non_banked code pointer modifier.

I/O ACCESS

The program may access the Z80/64180 I/O system using the following data type:

```
sfr
```

NON-VOLATILE RAM

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

```
no_init
```

CALLING MECHANISMS

By default the mechanism by which the compiler calls a function is determined by the memory model chosen. The program may achieve additional efficiency for special cases by overriding the default using one of the function modifiers:

```
interrupt    monitor    using_____C_task
```

#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 bitfie1ds=default
```

EXTENSION CONTROL

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

FUNCTION ATTRIBUTE

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

MEMORY USAGE

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

WARNING MESSAGE CONTROL

```
#pragma warnings=on
#pragma warnings=off
#pragma warnings=default
```

INTRINSIC FUNCTION SUMMARY

Intrinsic functions allow very low-level control of the Z80/64180 microprocessor. To use them in a C application, include the header file intrz80.h. The intrinsic functions compile into in-line code, either a single instruction or a short sequence of instructions.

For details concerning the effects of the intrinsic functions, see the appropriate microprocessor documentation.

GENERAL

halt

INTERRUPTS

enable_interrupt
disable_interrupt
interrupt_mode_0
interrupt_mode_1
interrupt_mode_2
load_I_register
dump_I_register

MEMORY AND I/O

input
output
_opc
input_block_inc
input_block_dec
output_block_inc
output_block_dec
input8
output8
address_24_of

Z8018X/64180 ONLY FUNCTIONS

sleep
output_memory_block_inc
output_memory_block_dec

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.

_C_task

Declares a function that does not restore registers.

SYNTAX

__C_task *declarator*

DESCRIPTION

Functions declared using this keyword do not restore registers. It can be used with ma i n () and process-main functions.

EXAMPLES

The example below specifies that ma i n () does not restore registers.

```
__C_task  main()
{
   . . .
}
```

banked

Declares a banked function.

SYNTAX

storage-class banked *declarator*

DESCRIPTION

In the large memory model, the default position for functions is within the single databank. The banked keyword indicates that the function is in a different bank, and so that the compiler must call it by the slower, banked method.

EXAMPLES

The function my_f unc is compiled using the banked memory model, but specified as located in the non-banked area. The non-banked function calls an assembly routine, my_moni tor, which is located in the banked area. A banked call is therefore required:

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

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

The keyword banked for my_moni tor is not actually needed since banked is the default. Including the keyword improves readability and may aid maintenance.

interrupt

Declare interrupt function.

SYNTAX

storage-class `interrupt` *function-declarator*
storage-class `interrupt` *[vector]* *function-declarator*
storage-class **`interrupt`** ***[vector]*** **`using`**
`[ALTERNATE_SET|JP_TO_HANDLER|NMI_INTERRUPT]`
fundi *on-declarator*

PARAMETERS

function-declarator	A void function declarator having no arguments.
`[vector]`	A square-bracketed constant expression yielding the vector address as an offset from the beginning of the INTV EC table.
`[ALTERNATE_SET]`	A square-bracketed string specifying that the alternate register set is to be used
`[JP_TO_HANDLER]`	A square-bracketed string specifying that the interrupting device will place an RST instruction on the data bus.
`[NMI_INTERRUPT]`	A square-bracketed string specifying a non-maskable interrupt.

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. 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.

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.

EXAMPLES

The Z80 has three different interrupt modes. This results in three different ways of calling an interrupt function.

The example below assumes that the interrupting device places an RST instruction on the data bus. If INIVEC is set to 8, the code below will result in a JP instruction to the address of ext_1 being inserted at address 8 of the table.

```
interrupt [0] using [JP_TO_HANDLER] void ext_1(void)
{
  intvar++;
}
```

The example below shows how to code an interrupt function which uses a vectored 16-bit interrupt or a direct call.

```
interrupt [0x24] void ext_2()      /* handler for external
                                   interrupt 0 */
{
  output8(33,6);
}
```

A vectored call is composed of the I register and the 8 bits on the data bus. The address of ext_2 will be placed into the INIVEC table at offset 24. The I register must be set in CSTARTUP or by using the load_I_register() intrinsic. The interrupting device must know the low byte corresponding to location 24 of the INIVEC table.

A direct call must use the 16-bit address of the interrupt function.

The example below does not install a vector in the INIVEC table. Either the interrupting device must supply a call and an address, or the address of

the function must be explicitly installed by the application to an interrupt table.

```
interrupt void titner_A0()          /* handler for timer A0
                                    interrupt */
{
  if (input8(33)  start_engine();
}
```

The assembler code below shows how to directly install an interrupt vector for mode 2:

```
RSEG          INTVEC
EXTERN         timer_A0
DEFS          4                     ;skip 4 bytes
DEFW           timer_A0
```

The assembler code below shows how to directly install an interrupt instruction for modes 0 and 1:

```
RSEG          INTVEC
EXTERN        my_int
DEFS          16-SFBCINTVEC)       ; skip to address 16
JP            my_int
```

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 moni tor is equivalent to a normal function in all other respects.

EXAMPLES

The example below disables maskable interrupts while the flag variable is modified.

```
char printer_free;                  /* printer-free semaphore */
monitor int got_f!ag(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 ....
}
```

nonjbanked

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 the compiler can call it by the faster, unbanked method.

EXAMPLES

Function my_nb has been coded as non-banked in another module. The module containing my_nb_cal 1 er was compiled with the banked memory option:

```
extern non_banked void my_nb(void)
{
  ...
}
void my_nb_ca Her (void)
{
  ...
  my_nb();                          /* 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_i nit 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_i nit 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 1-68.

EXAMPLES

The examples below show valid and invalid uses of the no_init keyword.

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

sfr

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

SYNTAX

`sfr` *identifier* = *constant-expression;*

DESCRIPTION

`sfr` denotes a Z80/64180 SFR-register which:

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

The value of an `sfr` variable is the contents of the SFR register at the address *constant-express ion.* All operators that apply to integral types except the unary & (address) operator maybe applied to `sfr` variables.

Predefined `sfr` declarations for popular members of the Z80 family are supplied; see *Installed files,* page 1-9.

The `sfr` type generates an IN or OUT instruction for assignments (I NO or OUTO for the 64180 and Z8018X).

EXAMPLES

The example below shows how to define STATO.

```
sfr STATO = 0x4;                    /* Defines P0 */
void func()
{
  STAT0=1;                          /* Set entire port STAT0=
                                       00000001 */
  if (STATO & 8) printfCW);         /* Read entire STATO and
                                       mask bit 3 */
}
```

using

Specifies a register bank for use by an interrupt service routine.

SYNTAX

{storage-class} interrupt *{vector}* {using *[flags]* } function-declarator

PARAMETERS

vector — A constant expression which is the offset from the start of INTVEC where the vector or the J P is to be installed.

flags — A number which selects options to be used when processing the interrupt. The file i ntz80. h contains the following definitions:

ALTERNATE_SET	The interrupt saves the environment using EXX and EX AF.AF"
NMI_INTERRUPT	The return from the interrupt is done with a RETN instruction instead of the default RET I.

JP_TO_HANDLER A JP label is inserted instead of the DEFW label in the interrupt table.

DESCRIPTION

The using keyword allows an interrupt service routine declaration to specify details about the interrupt type (NMI or IRQ), how to call the interrupt function, and whether the alternate register set should be used.

EXAMPLES

The example below assumes that the interrupting device places an RST instruction on the data bus. If INTVEC is set to 8, the code below will result in a JP instruction to the address of ext_1 being inserted at address 8 of the table.

```
interrupt [0] using [JP_TO_HANDLER] void ext_1(void)
{
  intvar++;
}
```

The example below shows how to code an interrupt function which uses a vectored 16-bit interrupt.

```
interrupt [0x24] void ext_2()     /* handler for external
                                     interrupt 0 */
{
  output8(15,6);
}
```

A vectored call is composed of the I register and the 8 bits on the data bus. The address of ext_2 will be placed into the INTVEC table at offset 24. The I register must be set in CSTARTUP or by using the load_I_register() intrinsic. The interrupting device must know the low byte corresponding to location 24 of the INTVEC table.

The example below does not install a vector in the INTVEC table. Either the interrupting device must supply a call and an address, or the address of the

function must be explicitly installed by the application to an interrupt table.

```
interrupt void timer_A0()          /* handler for timer AO
                                   interrupt */
{
  if (input8(33) start_engine();
}
```

The usi ng flags can be combined as shown below:

```
interrupt [8] using [ALTERNATE_SET|JP_TO_HANDLER]
my_int(void)
{
  count++;
}
```

#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

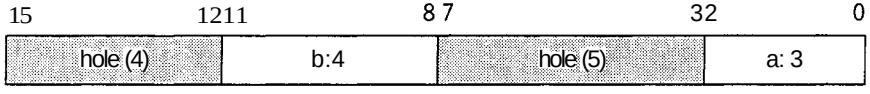
This directive causes the compiler to allocate bitfields starting at the most significant bit of the field, instead of at the least significant bit. The ANSI standard allows the storage order to be implementation-dependent, so you may run into portability problems, which this keyword can be used to avoid.

EXAMPLES

The default layout of

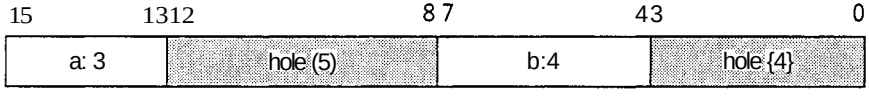
```
struct
{
 short a:3;                    /* a is 3 bits */
 short :5;                     /* this reserves a hole of
                                  5 bits */
 short b:4;                    /* b is 4 bits */
} bits;                        /* bits is 16 bits */
```

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:



function =__C_task

Declares a function that does not restore registers on exit.

SYNTAX

```
#pragma function=__C_task
```

DESCRIPTION

Functions declared with this pragma do not restore registers. It can be used with main() and process-main functions.

EXAMPLES

The example below specifies that ma i n () does not restore registers.

```
#pragma function-__C_task
main()
{
  . . .
}
```

function = default

Restores function definitions to the default type.

SYNTAX

```
#pragma function=defau1t
```

DESCRIPTION

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

EXAMPLES

The example below specifies that an external function f 1 can be called as a non_banked function, while f3 is the default type (banked or non_banked depending on the compiler options).

```
#pragma function=banked
extern void fl();              /* Identical to extern far
                                  void fl() */
#pragma function=defau!t
extern int f3();               /* Default function type
                                  */
```

function = interrupt

Makes function definitions interrupt.

SYNTAX

#pragma function=interrupt

DESCRIPTION

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

See the file i ntz80. h for a list of pre-defined interrupt function addresses. The definitions assume that the interrupt address segment INTVEC is located at 0x08.

Note that #pragma f uncti on=i nterrupt does not offer a vector option.

EXAMPLES

The example below shows an interrupt function process_int. The address of this function must be placed into the I NT V EC table.

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

function = monitor

Makes function definitions moni tor.

SYNTAX

```
#pragma function=monitor
```

DESCRIPTION

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

EXAMPLES

The example below disables interrupts while a flag bit is set.

```
#pragma function=monitor
void f2()                             /* Will make f2 a monitor
                                      function */
{
  flag |= 01;
}
```

function = nonjbanked

Makes function definitions non_banked.

SYNTAX

#pragma function=non_banked

DESCRIPTION

This directive places subsequent function definitions into the non-banked code area. It is an alternative to the function attribute non_banked.

EXAMPLES

The example below declares f 1 to be in the non-banked memory area. The external function should be used with an accessible segment identifier such as RCODE. A function declared as non-banked will appear in the RCODE segment.

```
#pragma function=non_banked
extern void fl();                /* Identical to extern
                                 non_banked void fl() */
```

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 1 anguage=extended.

language = extended

Makes extended kejrwords available.

SYNTAX

#pragma language=extended

DESCRIPTION

This directive makes the extended kejrwords available regardless of the state of the -e compiler option. It is an alternative to the -e compiler option. See *Extended keyword reference,* page 1-85, for details.

EXAMPLE

In the example below, the non_banked extended language modifier is enabled for the definition of the function ccount. mycount is defined in the standard way.

```
#pragma language=extended
extern non_banked int ccount(void)
#pragma language=default
extern int mycount(void)
```

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 1-117.

EXAMPLES

The example below places the constant array arr into the ROM segment TABLE.

```
#pragma memory=constseg(TABLE)
char ar[] = {6, 9. 2. -5, 0};
#pragma memory = default
```

If another module accesses the array it must use an equivalent declaration:

```
#pragma memory=constseg(TABLE)
extern char * arr;
```

memory = dataseg

Directs variables to the named segment by default.

SYNTAX

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

DESCRIPTION

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

No initial values maybe 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.

EXAMPLES

The example below causes four bytes to be allocated from the named segment myseg.

```
#pragma memory=dataseg (myseg)
char myseg_cl;
char myseg_c2;
int myseg_int;
#pragma memory=default
```

If another module wishes to access these symbols, the equivalent extern declaration should be used:

```
#pragma memory=dataseg(myseg)
extern char myseg_cl;
```

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.

EXAMPLES

See *Memory = dataseg*, page 1-104.

memory = nojbtiit

Direct variables to the N0_I NIT segment by default.

SYNTAX

```
#pragma memory=no_init
```

DESCRIPTION

This directive directs variables to the no_i nit segment, so that they will not be initialized and will reside in non-volatile RAM. It is an alternative to the memory attribute no_i n i t. The default may be overridden by the memory attributes.

The no_i n i t segment must be linked to coincide with the physical address of non-volatile RAM; see *Configuration*, page 1-59, for details.

EXAMPLES

In the example below the variable array buffer is not initialized when the program starts.

```
#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.

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 warm'ngs=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.

INTRINSIC FUNCTION REFERENCE

This chapter describes the intrinsic functions in alphabetical order.

address_24_of

Returns the address of a function or data area in the banked area.

SYNTAX

address_24_of*{memory_1 ocation);*

PARAMETERS

memory- location A pointer to a memory location in a banked code segment.

DESCRIPTION

The 24-bit address can be used by intrinsic functions.

EXAMPLE

The example below creates an area of banked data and copies data from the banked area into a buffer in RAM.

```
extern non_banked void memcpy_from_bank (char *, unsigned
int, unsigned long);
#pragma memory=constseg(BANKED_DATA)
const unsigned char bank_data[]={l,2,3,4,5};
#pragma memory=default
unsigned char buf[10]
void main(void)
```

```
{
  unsigned char str[16];
  for (i=0; i<5; i+=2)
  memcpy_from_bank(buf,2,address_24_of(&bank_data)+i);
}
```

The function which does the copying must change the bank pointers. It must therefore be located in non-banked memory. It can only access the following intrinsic library functions: memcpy, strcpy, strcat, strlen, memset, memcmp, strump, or strchr. Other library functions will fail to access the banked-data memory as the function-call mechanism switches the bank.

```
#include <stdio.h>
#include <string.h>
#inc"lude "intrz80.h"

sfr bank_port=39;

non_banked void memcpy_from_bank( char *dest, unsigned int
count, unsigned long address)
{
  char bankvar=bank_port;         /* save old bank pointer
                                  */
  /* switch memory banks to the data bank */
  bank_port=address>>16;
  /* test for space to copy string */
  if (str!en(char *) address <max)
    strcpy(dest, (char *) address);
  else
    dest=NULL;
  /* restore bank */
  bank_port=bankvar;
  return dest;
}
```

disable_interrupt

Generates a DI instruction.

SYNTAX

```
void disable_interrupt(void);
```

dump_I_register

Reads from the I register and returns the contents.

SYNTAX

```
unsigned char dump_I_register(void);
```

enable_interrupt

Generates an EI instruction.

SYNTAX

```
void enable_interrupt(void);
```

halt

Generates a HALT instruction.

SYNTAX

```
void halt(void);
```

input

Reads from a port using a 16-bit address IN instruction.

SYNTAX

```
unsigned char input(unsigned short)
```

input8

Reads from a port using the I NO instruction for the 64180 or the 8-bit address IN instruction on the Z80.

SYNTAX

```
unsigned char input8(unsigned char)
```

DESCRIPTION

For the 64180, the port number must be resolvable at compile time. The Z80 does not have this restriction.

input_block_dec
input_block_inc

Generates an INIRor INDRinstruction.

SYNTAX

```
void input_b1ock_dec(unsigned char, unsigned char *,
unsigned char);
void input_block_inc(unsigned char, unsigned char *,
unsigned char);
```

DESCRIPTION

Generates anINDRorlNIR instruction. The first argument is the port (C), second is the memory address (HL), and the third argument is the count (B).

interrupt_mode_0
interrupt_mode_l
interrupt_mode_2

Generates an IMO, IMl, or IM2 instruction, respectively.

SYNTAX

```
void interrupt_mode_0(void);
void interrupt_mode_l(void);
void interrupt_mode_2(void);
```

output

Writes to a port using the OUT instruction with a 16-bit port address.

SYNTAX

```
void output(unsigned short, unsigned char);
```

DESCRIPTION

The first argument is the port and the second the value to be written.

output8

Writes to a port using the OUTO instruction for the 64180 or the OUT instruction for the Z80.

SYNTAX

```
void output8(unsigned char, unsigned char);
```

DESCRIPTION

The first argument is the port number and the second is the value to be written.

For the 64180, the port number must be resolvable at compile time. The Z80 does not have this restriction.

output_block_dec
output_block_inc

Generates an OTDR or OTIR instruction.

SYNTAX

```
void output_b1ock_dec(unsigned char, unsigned char *,
unsigned char);
void output_block_inc(unsigned char, unsigned char *,
unsigned char);
```

DESCRIPTION

The first argument is the port (C), second is the memory address (HL), and the third argument is the count (B).

output_memory_block_dec
output_memory_block_inc

Generates an OTDMR or OTIMR instruction (64180/Z8018X only).

SYNTAX

```
void output_memory_block_dec(unsigned char, unsigned char *,
unsigned char);
void output_memory_block_inc(unsigned char, unsigned char *,
unsigned char);
```

DESCRIPTION

The first argument is the port, second is the memory address, and third argument is the count.

sleep

Generates an SLP instruction (64180/Z8018X only).

SYNTAX

```
void sleep(void)
```

_opc

Inserts an opcode.

SYNTAX

```
_opc(c)
```

DESCRIPTION

The _opc() intrinsic takes a single constant character c as a parameter; this is emitted by the compiler in the form of a DEFB assembler command. The intention of the macro is to create assembler opcodes for instructions difficult to describe in C.

To use this macro you must include the line #include <intrz80.h> and select the extended language option -e either from the command line or with the corresponding #pragma.

Use this function with great caution as it can easily confuse the optimizer.

EXAMPLE

The example below increments the 16-bit contents of a location in memory. Note that (*((short*)0x0020))++ will produce the same code from C.

```
void set_fixed_location()

{
  _opc(0x2A);          /* load HL */
  _opc(0x20);          /* address */
  _opc(0x00);
  _opc(0x23);          /* increment HL */
  _opc(0x22);          /* store value */
  _opc(0x20);
  _opc(0x00);
}
```

Standard C statements can almost always be used instead of _opc and will allow the compiler to optimize the code. The use of _opc is discouraged.

ASSEMBLY LANGUAGE INTERFACE

The ICCZ80/64180 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:

```
iccz80 shell -A -q -L
```

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

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

The following sections describe the interface in detail.

CALLING CONVENTION

Up to two parameters can be passed in registers; other parameters are transferred on the stack.

The compiler assembler interface selects the parameters that can be placed in the registers as follows:

Parameters, types, and locations

1	*2*	*Remaining parameters*
Byte E	Byte C	All types Pushed
Byte E	Word BC	All types Pushed
Byte E	3 bytes (pointer) Pushed	All types Pushed
Word DE	Byte C	All types R7
Word DE	Word BC	All types Pushed

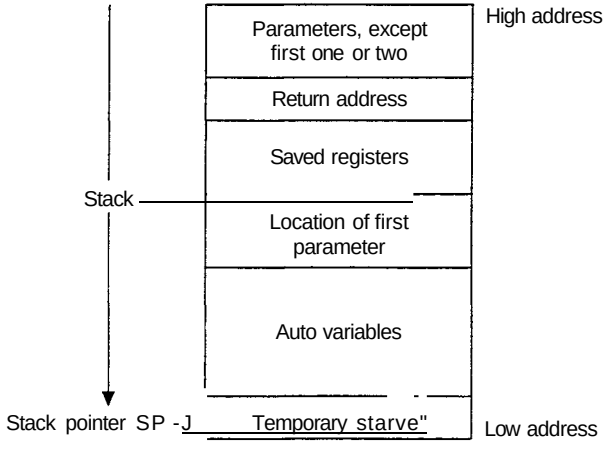
Parameters, types, and locations

1	*2*	*Remaining parameters*
3 bytes (pointer) CDE	All types Pushed	All types Pushed
4 bytes (long etc.) BCDE	All types Pushed	All types Pushed
Variable arguments Pushed	All types Pushed	All types Pushed

STACK USE

The remaining parameters, which are not transferred in registers, are pushed on the stack in reverse order, ie the last parameter is transferred first. Pushed parameters are removed by the caller after returning from the called function.

The stack model used by C functions differs depending on the options for speed or monitor functions. A typical stack frame is shown below. If you do not use a shell function for the assembler routine, you must ensure that your stack usage is compatible with the compile options:



VARIABLE ARGUMENTS

All va ra rg functions expect every parameter on the stack. Creating a va ra rg function in one module and calling it from another without the prototype will fail:

```
/* In module 1 */
void foo(int xx, _.)
{
  …
}
----------------------------
/* In module 2 */
main()
{
  food,2,3,4);                  /* Will fail if prototype
                                for foo() has not been
                                included */
}
```

You will get an error message from the linker if you have compiled with the -gA option. Library vararg functions such as pri ntf are recognized by the compiler and an error will be given if there is no prototype defined.

If you create a routine in assembler directly, you will need to manage the stack. It is much simpler to create a C shell with the correct number and type of parameters and then modify the assembler output.

RETURN VALUES

The return value is given in registers if possible, otherwise it is at the pointed-to location in the caller's own storage space.

The return value from a function will be placed into the registers as shown opposite.

Type	*Register*
char	A (from a non-banked function)
char	L (from a banked function)
word	HL
pointer	HL
banked function pointer	CHL
long, float, or double	BCHL

PRESERVING REGISTERS

A and HL are always considered destroyed after a function call. Registers used for parameters are also considered destroyed after a function call; this includes the entire 16-bit register. For example, if E is used for a parameter DE is considered destroyed. All other registers (excluding return value registers) must be preserved by the called function.

For example, consider the following function prototype:

```
long foo(int, ...)
```

A and HL are always destroyed, BC is destroyed since it is used for the return value. Since this is a vararg function it does not take any parameter in registers which means that IX, IY, and DE must be preserved as usual.

If the alternative register set is in use by the code generator -ua, HL' is considered destroyed after each function call, but DE' and BC' must be preserved.

CALLING ASSEMBLY ROUTINES FROMC

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

- Conform to the calling convention described above.
- HaveaPUBLIC entry-point label.
- Be prototyped before any call, to allow type checking and promotion of parameters, as in extern int food nt i, int j).

In addition, it should be located in the segment CODE or, if declared with the non_banked keyword, RCODE (but you can use any segment name provided you declare it properly in the link file).

LOCAL STORAGE ALLOCATION

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

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

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 routine must exit using RETI or RETN.

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 user must place the actual interrupt routine in the RCODE segment. See the interrupt keyword in the *Extended keyword reference*, page 1-85, for an example of interrupt installation.

EXAMPLE

A shell function declared as long my_int (long a, unsigned short b) for Z80/64180 is shown below:

```
long my_int (long a, unsigned short b)
{
  long temp;
  if (b>6) temp—1;
  else temp=a/b;
  return(temp);
}
```

Parts of the list file are described below:

```
     1          long my_int (long a, unsigned short b)
     2          {
\    0000            my_int:
\    0000  CD0000            CALL    ?ENT_AUT0_DIRECT_L09
\    0003  FCFF              DEFW    -4
```

The function is entered and temporary variable space is reserved.

```
     3            long temp;
     4            if (b>6) temp=-l;
\    0005  DD4E08            LD      C,(IX+8)
\    0008  DD4609            LD      B,(IX+9)
\    000B  210600            LD      HL.6
\    000E  A7                AND     A
\    000F  ED42              SBC     HL.BC
\    0011  3010              JR      NC,70001
```

The short parameter is tested and a jump made to the assignment.

```
\   0013            70000:
\   0013  06FF              LD      B.-l
\   0015  DD70FC            LD      (IX-4),B
\   0018  DD70FD            LD      (IX-3KB
\   001B  DD70FE            LD      (IX-2),B
\   001E  DD70FF            LD      (IX-l).B
\   0021  1826              JR      70002
```

The temporary variable pointed to by IX is set to -1.

```
\   0023            70001:
   5            else: temp=a/b;
\   0023  DD6E08            LD      L.CIX+8)
\   0026  DD6609            LD      H.CIX+9)
\   0029  010000            LD      BC.O
\   002C  C5                PUSH    BC
\   002D  E5                PUSH    HL
\   002E  DD4E04            LD      C.CIX+4)
\   0031  DD4605            LD      B.CIX+5)
\   0034  DD6E02            LD      L,(IX+2)
\   0037  DD6603            LD      H,(IX+3)
\   003A  CD0000            CALL    ?SL_DIV_L03
\   003D  DD75FC            LD      (IX-4),L
\   0040  DD74FD            LD      (IX-3),H
\   0043  DD71FE            LD      (IX-2),C
\   0046  DD70FF            LD      (IX-l).B
```

Note that the compiler has adjusted the parameter b by extending it to four bytes.

```
\   0049            70002:
   6            return(temp);
\   0049  DD4EFE            LD      C,(IX-2)
\   004C  DD46FF            LD      B.(IX-l)
\   004F  DD6EFC            LD      L.(IX-4)
\   0052  DD66FD            LD      H.UX-3)
```

The temporary variable is copied into the registers which hold a long return value.

```
      7          }
\     0055  C30000              JP        ?LEAVE_32_L09
\     0058                      END
```

The compiler uses an indirect exit to clean up after the function calls. If you had used a direct entry and exit, you would have needed to manage the temporary variables and stack maintenance yourself.

SEGMENT REFERENCE

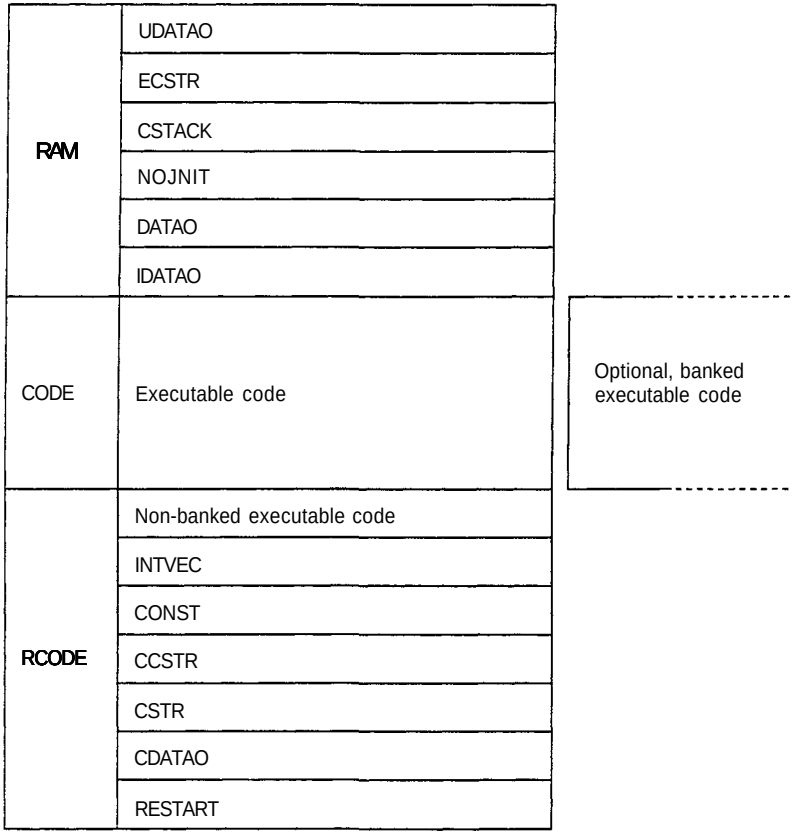
The IAR C Compiler places code and data in to named segments which are referred to by the linker. Details of the segments is required for programming assembly language modules, and is also useful when interpreting the assembly language output of the compiler.

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.

MEMORY MAP DIAGRAM

The diagram on the following page shows the Z80/64180 memory map, and the allocation of segments within each memory area.


RAM
UDATAO
ECSTR
CSTACK
NOJNIT
DATAO
IDATAO
CODE
Executable code
Optional, banked executable code
RCODE
Non-banked executable code
INTVEC
CONST
CCSTR
CSTR
CDATAO
RESTART

CCSTR

String initializers.

TYPE
Read-only.

DESCRIPTION
Assembly-accessible.

Holds C string literal initializers when the -y (put string literals into variable section) compiler option is active.

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.

This segment and length is normally defined in the XLINK file by the command:

```
-Z(DATA)CSTACK+/7/7=sta/~t
```

where *nn is* the length and *start* is the location.

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 *Command line options* in the *IAR C Compiler - General Features* guide.

ECSTR

Writable string literals.

TYPE
Read/write.

DESCRIPTION
Assembly-accessible.

Holds writable 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 *Command line options* in the *IAR C Compiler - General Features* guide.

CDATAO

Variable initializer.

TYPE
Read-only.

DESCRIPTION
Compiler-only.

Holds non-banked variable initializers. These values are copied over from CDATAO to IDATAOby CSTARTUP during initialization.

DATAO

Uninitialized near static variables.

TYPE
Read/write.

DESCRIPTION
Compiler-only.

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

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).

IDATAO

Initialized near static variables.

TYPE
Read/write.

DESCRIPTION
Compiler-only.

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

NOJNIT

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 n o_i n i t or created n o_i n i t by use of the memory #pragma, or created manually from assembly language source.

RCODE

Vector handling code.

TYPE
Read-only.

DESCRIPTION
Assembly-accessible non-banked code used by code generator intrinsic functions.

This segment can also be used for user-written non-banked assembler code that is not called from C (interrupt handlers and similar resident code).

TEMP

Static mode auto variables. Used with the -d option.

TYPE
Read/write.

DESCRIPTION
STATIC function arguments are always placed on the stack. Re-entrant mode (the default) sometimes generates more and slower code. Auto variables are allocated and deallocated dynamically (on the stack).

UDATAO

Uninitialized nea r static variables.

TYPE
Read/write.

DESCRIPTION
Assembler-accessible.

Holds near 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 CSTARTU P during initialization. UDATAO can also hold user-written data elements that should initially be set to zero.

Z80 COMMAND LINE OPTIONS

In addition to the command line options described in *Command line options* in the *IAR C Compiler - General Features* guide, the Z80/64180 C Compiler has the following options:

ICCZ80 COMMAND LINE OPTIONS SUMMARY

-m{l | b)	Selects memory model.
-u{a | r | u}	Sets optimization options.
-v{0 | 1}	Selects the processor.
-W[ss]	Set stack optimization: allow ss bytes of garbage.

The option - P (generate PROMable code) is ignored - the compiler generates PROMable code by default.

-m

Selects memory model.

SYNTAX

-m{l | b}

DESCRIPTION

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

Modifier	*Memory model*
1	Large (the default)
b	Banked

For more information see *Memory model*, page 1-61.

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

-u

Sets optimization options.

SYNTAX

-u{a | r | **u}**

DESCRIPTION

Sets the compiler optimization as follows:

Option	*Description*
a	Lets the compiler use the alternative register set. This excludes them from being used by interrupts.
r	Lets the compiler use RST vectors to reduce the code size. Banked call, return, and function routines will be called using RST instructions rather than CALL. This will reduce the code size, but will also slow the code down.
u	Lets the compiler use undocumented instructions to reduce code size and increase execution speed; see *Optimization*, page 1-69, for details.

-V

Selects the processor.

SYNTAX

-v{0 | 1}

DESCRIPTION

Use -vO to generate code for the Z80 processor and -vl for the 64180/Z8018X.

-W

Sets the stack optimization limit.

SYNTAX

-W[ss]

DESCRIPTION

The number of clean-ups of the stack is reduced by specifying stack optimization with the -W option. For example, by specifying -W50, the compiler is instructed to allow 50 bytes of garbage on the stack before triggering stack clean-up. The default setting is -W16, ie 16 bytes of garbage. Also see *Stack size,* page 1-69.

Z80 DIAGNOSTICS

In addition to the error and warning messages described in *Diagnostics* in the *IAR C Compiler - General Features* guide, the Z80/64180 C Compiler has the following error messages:

Error message	*Suggestion*
Address argument required	The address_24_of() intrinsic function must take a pointer to an absolute object.
non-banked/interrupt functions must be defined in a separate module	When compiling in banked memory model and a function is declared to be non_banked, all functions in that module must be non_banked. This error is also given for interrupt functions as they must reside in the RCODE segment.
Cannot have both '-ua' and ALTERNATE_SET interrupts	When compiling with the switch -ua no interrupt may be declared to use the alternative register set.
You must #include <stdio.h> if you call printf/scanf	Since ICCZ80 uses a special parameter model for varargs functions a prototype must be included. Include stdio.h to get the prototype.
Constant argument required	Some intrinsic functions (_opc, input8, output8) require an argument that can be evaluated at compile time.
Cannot compare banked code pointers	Comparing banked pointers using <, >, <=, and >= is forbidden.

Error message	*Suggestion*
Long bitfields are not supported	Only char, int, or short bitfields are supported.

IAR C COMPILER - GENERAL FEATURES

Third edition: July 1994
Part no: ICCGEN-3

CONTENTS

CONTENTS

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 - 1, the order of multiple options is important.

The options are arranged into the following functional groups:

FILE CONTROL

- a *file*	Generates assembler source.
-kprefix	Generates assembler source.
- f *file*	Reads command line options from a file.
- *G*	Opens the standard input as source.
-1 prefix	Adds an include file search prefix.

-1 *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.
-p/7	Formats listing into pages.
- q	Puts mnemonics in the listing.
-1 *n*	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.
- H *name*	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 c h a r.
- C	Enables nested comments.

-g[A][0]	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 maybe assembled by the appropriate IAR Assembler.

If the -1 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

-kprefix

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 leafhame but with the target-specific assembly source extension.

The -A option maybe 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 -1 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 ofcharasunsigned charorsignedcharis implementation dependent.

By default, the IAR C Compiler treats char as equivalent tounsigned 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 -D *symb=xx* option.

The -Dsym/?=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 maybe 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.xch*

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 maybe 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][0]
```

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 -gO 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 -gO[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 */
_________A

"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 */
_ ._____________________. A

"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:

```
}
A

"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;
  putsC'This code is unreachable");
exit:
 return 0;
}
```

Error:

```
  putsCThis 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  Warrn'ng[2]: Macro 'my_macor' is already #undef
```

Valid but ambiguous initializers
Program:

```
typedef struct tl {int fl; int f2;} typel;
typedef struct t2 {int f3; typel f4; typel 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 */
              ^
"arn'ndex.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

-Wname

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 -1 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 -1 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 -1 options on a single command line.

Note that the compiler simply adds the -1 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.

-1

Generates a listing.

SYNTAX

-1 *file*

DESCRIPTION

By default, the compiler does not generate a listing. Use the -1 option to generate a listing to the named file. The filename consists of a leafhame optionally preceded by a pathname and optionally followed by an extension. If no extension is given, . 1 st is used.

The -1 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 leafhame but with the extension .1st.

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 -1 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 -0.
- 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 -0 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 -0 to specify a prefix which the compiler adds to the leafhame, allowing the object code to redirected to an alternative directory.

The -o and -0 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	#i n c 1 u d e 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 #i ncl ude 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
-to

DESCRIPTION

By default, the listing is formatted with a tab spacing of 8 characters. Use -1 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 #i f structures, are listed. Use -T to list active lines only.

-u

Undefines a symbol.

SYNTAX

-\isymb

DESCRIPTION

- U *symb* 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 h h: mm: s s 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 (* signaHint___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.

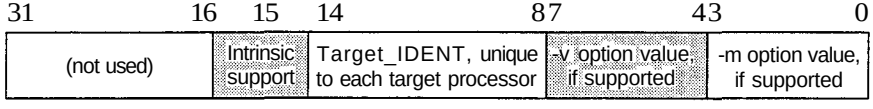
COMPILER VERSION

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

_a rgs $ 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 _a rgt$ format.
1	Parameter 1 size in bytes.
2	Parameter 2typein_argt$ format.
3	Parameter 2 size in bytes.
2 n - 2	Parameter ntypein_argt$ format.
2n -1	Parameter n size in bytes.
2n	\0

Sizes greater than 127 are reported as 127.

_a rgs $ 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 si zeof operator cannot be used in #i f and #el i f 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 #i ncl ude 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.

LD3RARY 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 isalphad'nt c)`	Letter equality.
`iscntrl`	`int iscntrl(int c)`	Control code equality.
`isdigit`	`int isdigitd'nt c)`	Digit equality.
`isgraph`	`int isgraph(int c)`	Printable non-space character equality.
`i s1 owe r`	`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	int _formatted_read (const char **line, const char **format, va_list ap)	Reads formatted data.
_formatted_write	int_formatted_write (const char* format, void outputf (char, void *) , void *sp. va_list ap)	Formats and writes data.
_medium _read	int _formatted_read (const char **line, const char **format, va_list ap)	Reads formatted data excluding floating-point numbers.
_jnedium _write	int_formatted_write (const char* format, void outputf (char, void *) , void *sp, va_list ap)	Writes formatted data excluding floating-point numbers.
_small _write	int _formatted_write (const char* format, void outputf (char, void *) , void *sp, va_list ap)	Small formatted data 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 *argl*, double *argZ)*	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 *argl*, double *argZ)*	Floating-point remainder.
frexp	double frexp(double *argl*, int **argZ)*	Splits a floating-point number into two parts.
ldexp	double ldexpCdouble *argl*, int *argZ)*	Multiply by power of two.
log	double log(double *arg)*	Natural logarithm.
loglO	double loglO(double *arg)*	Base-10 logarithm.
modf	double modf(double *value*, double **iptr)*	Fractional and integer parts.
pow	double pow(double *argl*, double *argZ)*	Raises to the power.
sin	double sin(double *arg)*	Sine.
sinh	double sinh(double *arg)*	Hyperbolic sine.
sqrt	double sqrtCdouble *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)*	Longjump.
setjmp	int setjmp(jmp_buf *env)*	Setsjump.

VARIABLE ARGUMENTS - stdarg.h

va_arg	type va_arg(va_list *ap*, *node)*	Next argument in function call.
va_end	void va_end(va_list *ap)*	Ends reading function call arguments.
va_list	char *va_list[l]	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 atoi(const char *nptr)	Converts ASCII to 1 ong int.
calloc	void *calloc(size_t nelem, size_t el size)	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 labsdong int j)	Long absolute.
Idiv	ldiv_t ldivdong 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 *sl, const void *s2, size_t n)	Compares memory.
memcpy	void *memcpy(void *sl, const void *s2, size_t n)	Copies memory.
memmove	void *memmove(void *sl, const void *s2, size_t n)	Moves memory.
memset	void *memset(void *s, int c, size_t n)	Sets memory.
strcat	char *strcat(char *sl, 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 *sl, const char *s2)	Compares two strings.
strcol1	int strcoll(const char *sl, const char *s2)	Compares strings.
strcpy	char *strcpy(char *sl, const char *s2)	Copies string.
strcspn	size_t strcspn(const char *sl, const char *s2)	Spans excluded characters in string.
strlen	size_t strlen(const char *s)	Stringlength.
strncat	char *strncat(char *sl, const char *s2, size_t n)	Concatenates a specified number of characters with a string.

strncmp	int strncmp(const char *sl, const char *s2, size_t n)	Compares a specified number of characters with a string.
strncpy	char *strncpy(char *sl, const char *s2, size_t n)	Copies a specified number of characters from a string.
strpbrk	char *strpbrk(const char *sl, const char *s2)	Finds any one of specified characters in a string.
strrchr	char *strrchr(const char *s, int c) right	Finds character from of string.
strspn	size_t strspn(const char *sl, const char *s2)	Spans characters in a string.
strstr	char *strstr(const char *sl. 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 - errnch

No functions (various error return values).

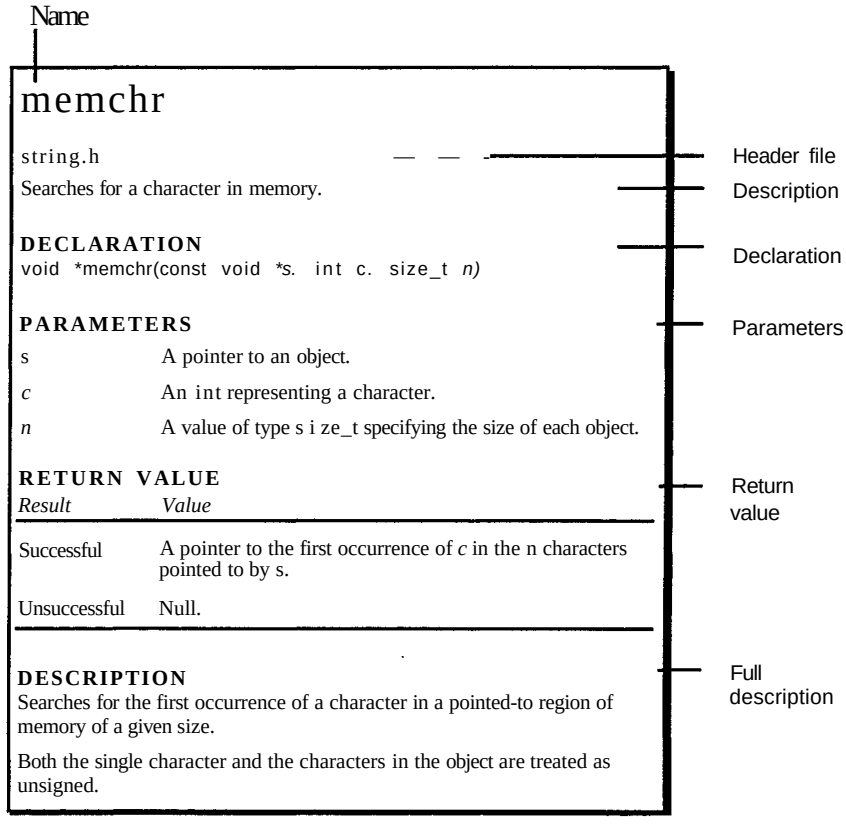
ASSERT - assert.h

assert	void assertdnt *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 doubl e 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 doubl e 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

express i on 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#defineNDEBUG statement before the #incl ude <assert.h> statement.

atan

math.h

Arc tangent.

DECLARATION

double atan(double *arg)*

PARAMETERS

arg A double value.

RETURN VALUE

The doublearctangentof *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 *argl,* double *argZ)*

PARAMETERS

argl	A double value.
arg2	A doubl e value.

RETURN VALUE

The doubl e arc tangent of *argl/argZ,* in the range [-pi, pi].

DESCRIPTION

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

atof

stdlib.h

Converts ASCII to doubl e.

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

"le-4" gives 0.0001

atoi

stdlib.h

Converts ASCII to i nt.

DECLARATION

int atoi(const char *nptr)

PARAMETERS

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

RETURN VALUE

The i nt 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 1 ong i nt.

DECLARATION

long atol(const char *nptr)

PARAMETERS

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

RETURN VALUE

The 1 ong 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 *cal1oc(size_t *nelem,* size_t *elsize)*

PARAMETERS

nelem	The number of objects.
elsize	A value of type s i ze_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 s i zeof 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 doubl e 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 doubl e value in radians.

RETURN VALUE
The doubl e cosine of *arg.*

DESCRIPTION
Computes the cosine of *arg* radians.

cosh

math.h

Hyperbolic cosine.

DECLARATION

double cosh(double *arg)*

PARAMETERS

arg Adouble value in radians.

RETURN VALUE

The doubl e 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 *numerhy* 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 exitdnt *status)*

PARAMETERS

status An i nt 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 doubl e 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 doubl e value.

RETURN VALUE
The double absolute value of *arg.*

DESCRIPTION
Computes the absolute value of the floating-point number *a rg.*

floor

math.h

Largest integer less than or equal.

DECLARATION

double floor(double *arg)*

PARAMETERS

arg A double value.

RETURN VALUE

A doub 1 e with the value of the largest integer less than or equal to *arg.*

DESCRIPTION

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

fitnod

math.h

Floating-point remainder.

DECLARATION

double fmod(double *argl,* double *arg2)*

PARAMETERS

argl	The doubl e numerator.
arg2	The doubl e denominator.

RETURN VALUE

The doubl e remainder of the division *argl/arg2.*

DESCRIPTION

Computes the remainder of *argl/arg2,* ie the value *argl - i*arg2,* for some integer i such that, if *a rg2* is non-zero, the result has the same sign as *argl* 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 mal 1 oc, cal 1 oc, or real 1 oc.

RETURN VALUE

None.

DESCRIPTION

Frees the memory used by the object pointed to by *ptr*. *ptr* must earlier have been assigned a value from mal 1 oc, cal 1 oc, or real 1 oc.

frexp

math.h

Splits a floating-point number into two parts.

DECLARATION

double frexp(double *argl,* int **arg2)*

PARAMETERS

argl	Floating-point number to be split.
arg2	Pointer to an integer to contain the exponent of *argl.*

RETURN VALUE

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

DESCRIPTION

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

The values are as follows:

mantissa * $2^{exponent}$ = 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 5.
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 isalnumCint *c)*

PARAMETERS

c An i nt representing a character.

RETURN VALUE

An i nt 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 isalphadnt c)
```

PARAMETERS

c An i nt representing a character.

RETURN VALUE

An i nt which is non-zero if cis 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 islowerdnt *c)*

PARAMETERS

c An i nt representing a character.

RETURN VALUE

An i nt which is non-zero if cis lower case, else zero.

DESCRIPTION

Tests whether a character is a lower case letter.

isprint

ctype.h

Printable character equality.

DECLARATION

int isprintdnt c)

PARAMETERS

c An int representing a character.

RETURN VALUE

An i nt 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 ispunctCint c)

PARAMETERS

c An i nt representing a character.

RETURN VALUE

An i nt which is non-zero if cis 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 isupperdnt c)

PARAMETERS

c An i nt representing a character.

RETURN VALUE

An i nt which is non-zero if cis upper case, else zero.

DESCRIPTION

Tests whether a character is an upper case letter.

isxdigit

ctype.h

Hex digit equality.

DECLARATION

int isxdigitO'nt c)

PARAMETERS

c An i nt representing a character.

RETURN VALUE

An i nt 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 1 ong 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` *argl*`,int` *arg2)*

PARAMETERS

argl	The doubl e multiplier value.
arg2	The i nt power value.

RETURN VALUE

The double value of *argl* 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 ldivdong int *numer,* long int *denom)*

PARAMETERS

numer	The long int numerator.
denom	The 1 ong int denominator.

RETURN VALUE

A structof type 1 di v_t holding the quotient and remainder of the division.

DESCRIPTION

Divides the numerator *numer by* the denominator *denom.* The type 1 di v_t is defined in std 1 i b. 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.

loglO

math.h

Base-10 logarithm.

DECLARATION
double loglOCdouble *arg)*

PARAMETERS
arg A doubl e 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 set jmp.
va 1	The i n t 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 *va* 7.

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 memcmpCconst void *sl, const void *s2, size_t n)
```

PARAMETERS

si	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 *si* with the first *n* characters of the object pointed to by *s2:*

Return value	*Meaning*
>0	si < s2
=0	si = s2
<0	si < s2

DESCRIPTION

Compares the first *n* characters of two objects.

memcpy

string.h

Copies memory.

DECLARATION

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

PARAMETERS

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

RETURN VALUE

si.

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 *sl, const void *s2, size_t n)

PARAMETERS

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

RETURN VALUE

si.

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 *va 1 ue.*

DESCRIPTION

Computes the fractional and integer parts of *value.* The sign of both parts is the same as the sign of *va 1 ue.*

pow

math.h

Raises to the power.

DECLARATION

double pow(double *argl,* double *argZ)*

PARAMETERS

argl	The double number.
argZ	The doubl e power.

RETURN VALUE

argl 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.

forma t 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 *forma t* string to be evaluated, converted, and written.

The form of a conversion specification is as follows:

% [f7ags] [field_width~\ [.precision] [length_modifieri conversion

Items inside [ ] are optional.

Flags

The *flags* are as follows:

Flog	*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*
	octa 1	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 OX.
x	Non-zero values prefixed with 0 X.	
0	Zero padding to field width (for d, i, o, u, x, X, e, E, f, g, and G specifiers).	

Field width

The fi el d_wi dth 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 precisionis 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 7 *ength_modi tier* is as follows:

Lengih_modifier	*Use*
h	before d, i, u, x, X, or o specifiers to denote a short int or unsigned short int value.
1	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 *con vers i on* 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. d d d E+d d.
f	Double value in the style [-]ddd.ddd.
g	Double value in the style of f or e, whichever is the more appropriate.
G	Double value in the style of F or E, whichever is the more appropriate.

Conversion	*Result*
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 intwrit.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; A represents space:

Statement	*Output*	*Number of characters output*
printf("%c",p[l])	B	1
printf("%d",i)	6	1
printf C%3d",i)	AA6	3
printf("%.3d",i)	006	3
printf("%-10.3d",i)	OOfiAAAAAAA	10
printf("%10.3d",i)	AAAAAAAOOfi	10
printf("Value=%+3d",i)	Value=A+6	9
printf("%10.*d",i,j)	AAA-000006	10
printf("String=[%s]",p)	String=[ABC]	12
printf("Value=%lX",l)	Value=186A0	11
printf("%f",fl)	0.000000	8
printf("%f",f2)	750000.000000	13
printf("%e",fl)	1.000000e-07	12
printf("%16e",d)	AAAA?.?nnnnnp+nn	16
printf("%.4e",d)	2.2000e+00	10
printf("%g",fl)	le-07	5
printf("*g",f2)	750000	6
printf("%g",d)	2.2	3

putchar

stdio.h

Puts character.

DECLARATION

int putcharCint *value)*

PARAMETERS

va 1 ue The i nt 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 pri ntf.

puts

stdio.h

Puts string.

DECLARATION

```
int putsCconst 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 i nt 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 s rand 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.orn	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 (On) or hexadecimal (Oxn, OXn).
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	Oneorfield_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*
[^A]	Any number of characters not matching] or any of the characters before the further, terminating]. For example, [^A]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 1 ongjmp causes execution to continue as if it was a return from setjmp, in which case the value of the i nt value given in the 1 ongjmp is returned.

DESCRIPTION

Saves the environment in *en v* for later use by 1 o n g j mp.

Note that setjmp must always be used in the same function or at a higher nesting level than the corresponding call to 1 ongjmp.

sin

math.h

Sine.

DECLARATION

double sin(double *arg)*

PARAMETERS

arg Adouble value in radians.

RETURN VALUE

The doubl e sine of *arg.*

DESCRIPTION

Computes the sine of a number.

sinh

math.h

Hyperbolic sine.

DECLARATION

double sinh(double *arg)*

PARAMETERS

arg A doubl e value in radians.

RETURN VALUE

The doubl e 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.
forma t	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 pri ntf except the output is directed to a string. See pri ntf for details.

sprintf does not use the function putcha r, and therefore can be used even ifputcharis 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 getcharis 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 *sl, const char *s2)

PARAMETERS

si	A pointer to the first string.
s2	A pointer to the second string.

RETURN VALUE

si.

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 *sl, const char *s2)
```

PARAMETERS

si	A pointer to the first string.
s2	A pointer to the second string.

RETURN VALUE

The i nt result of comparing the two strings:

Return value	*Meaning*
>0	si < s2
=0	si = s2
<0	si < s2

DESCRIPTION

Compares the two strings.

strcoll

string.h

Compares strings.

DECLARATION

```
int strcol1(const char *sl, const char *s2)
```

PARAMETERS

si	A pointer to the first string.
s2	A pointer to the second string.

RETURN VALUE

The i nt result of comparing the two strings:

Return value	*Meaning*
>0	si < s2
=0	si = s2
<0	si < 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 *sl, const char *s2)

PARAMETERS

si	A pointer to the destination object.
s2	A pointer to the source string.

RETURN VALUE

si.

DESCRIPTION

Copies a string into an object.

strcspn

string.h

Spans excluded characters in string.

DECLARATION

```
size_t strcspn(const char *sl, const char *s2)
```

PARAMETERS

si	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 *si* 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 s i ze_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 strncmpCconst char *sl, const char *s2, size_t n)

PARAMETERS

si	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	si < s2
=0	si = s2
<0	si < 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

si	*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

si.

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

si	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 *sl, const char *s2)
```

PARAMETERS

si	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 *si* 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 *sl, const char *s2)

PARAMETERS

si	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 5 J 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, *si* if s*2* 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 doubl e.

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 doubl e, 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 1 ong i nt 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 Ox 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 i nt value specifying the base.

RETURN VALUE

Result	*Value*
Successful	The unsignedlongint 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 i nt 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 Ox portion of a hex integer is allowed as the initial sequence.

tan

math.h

Tangent.

DECLARATION
double tan(double *arg)*

PARAMETERS
arg Adouble value in radians.

RETURN VALUE
The doubl e tangent of *arg.*

DESCRIPTION
Computes the tangent of arg radians.

tanh

math.h

Hyperbolic tangent.

DECLARATION
double tanh(double *arg)*

PARAMETERS
arg A doubl e value in radians.

RETURN VALUE
The double hyperbolic tangent of *arg.*

DESCRIPTION
Computes the hyperbolic tangent of *a rg* radians.

tolower

ctype.h

Converts to lower case.

DECLARATION

int tolowerdnt c)

PARAMETERS

c The i nt representation of a character.

RETURN VALUE

The i nt 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 i nt representation of a character.

RETURN VALUE

The i nt 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 paramN. 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_l i s t 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_sta rt that initialized va_list ap.

va_list

stdarg.h

Argument list type.

DECLARATION

```
char *va_list[l]
```

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 v a_l i s t 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_l i st 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 _f ormatted_read requires the special ANSI-defined macros in the file stda rg. h, described above. In particular:

There must be a variable *ap* of type v a_l i s t.

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
Cchar, void *;, void *sp, va_list ap)
```

PARAMETERS

format	A pointer to standard pri ntf/spri ntf format specification string.
outputf	A function pointer to a routine that actually writes a single character created by _f ormatted_wr i te. 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_wri te.
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 (voi d *) 0 as well as declared in the output function.
ap	A pointer of type va_l i s t to the variable-argument list.

RETURN VALUE

The number of characters written.

DESCRIPTION

Formats write data. This function is the basic formatter of pri ntf and spri ntf, 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_wri te is concurrently reusable (reentrant).

Note that the use of _f ormatted_wri te requires the special ANSI-defined macros in the file stdarg. h, described above. In particular:

- There must be a variable apof type va_l i st.
- There must be a call to va_start before calling _f ormatted_wri te.
- There must be a call to va_en d before leaving the current context.
- The argument tova_start must be the formal parameter immediately to the left of the variable argument list (_.).

For an example of how to use _f ormatted_wri te, see the file pri ntf. c.

_medium_read

icclbutl.h

Reads formatted data excluding floating-point numbers.

DECLARATION

```
int _medium_read (const char **line, const char **format,
va_1ist 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 v a_l i s t to the variable argument list.

RETURN VALUE

The number of successful conversions.

DESCRIPTION

A reduced version of _f ormatted_read which is half the size, but does not support floating-point numbers.

For further information see _f ormatted_read.

_medium_write

icclbutl.h

Writes formatted data excluding floating-point numbers.

DECLARATION

int _medium_write (const char ^*format,* void *outputf*Ychar, void *;, void *sp,* va_list *ap)*

PARAMETERS

fo rma t	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_wri te. 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_wri te.
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_l i s t to the variable-argument list.

RETURN VALUE

The number of characters written.

DESCRIPTION

A reduced version of_formatted_wri te which is half the size, but does not support floating-point numbers.

For further information see _f ormatted_wri te.

_small_write

icclbutl.h

Small formatted data write routine.

DECLARATION

```
int _sma11_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 _f ormatted_wri te. 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_wri te.
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 he specified with (voi d *) 0 as well as declared in the output function.
ap	A pointer of type va_J i st to the variable-argument list.

RETURN VALUE

The number of characters written.

DESCRIPTION

A small version of _f ormatted_wr i te which is about a quarter of the size, and uses only about 15 bytes of RAM.

The _small_wri te 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 _f ormatted_wri te.

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 vol ati 1 e, an attribute indicating that the object maybe modified by hardware and hence any access should not be removed by optimization.

SIGNED KEYWORD

ANSI C adds si gned, an attribute indicating that an integer type is signed. It is the counterpart ofunsignedand can be used before any integer type-specifier.

VOID KEYWORD

ANSI C adds voi d, 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 j 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 (namedform)	`extern intdong int val);`
ANSI (unnamedform)	`extern intdong 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 f 1 oat).

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 "\xlb\x43" /* Escape 'C \0 */
```

\x43 represents ASCII C which, if included directly, would be interpreted as part of the he'xadecimal constant.

STRUCTURE AND UNION ASSIGNMENTS

In K&R C, functions and the assignment operator may have arguments that are pointers to struct or uni on objects, but not struct or uni on objects themselves.

ANSI C allows functions and the assignment operator to have arguments that are struct or uni on 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 #e! i f directive allows more compact nested else-if structures.

```
#elif expression
...
is equivalent to:
#else
#if expression
...
#endif
```

#error

The #e r r o r 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 Ox or OX 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 (#def i ne 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 #defi ne statement was repeated.
19	Unmatched #else, #endif or #elif	Fatal.A#if, #ifdef,or #i fndef 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 #defi ne 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 #def i ne 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 #i nclude 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 #i ncl ude files.
35	Invalid declaration of "struct", "union" or "enum" type	A struct, union, or enumwas followed by an invalid token (s).
36	Tag identifier 'name' redeclared	A struct, uni on, 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 uni on 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 'C 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 0100)	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 en um type with type undefined or a reference to undefined struct or uni on type in a function prototype or as a si zeof argument.
68	"case" outside "switch"	There was a case without any active swi tch statement.
69	"interrupt" function may not be referred or called	Aninterrupt 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 def aul t without any active swi tch statement.
72	Multiple "default" within "switch"	More than one def aul t in one swi tch 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
	`struct`
	`{`
	`int *f:6; /* Invalid */`
	`}`
		There is a pointer to a field member of struct or union:

79	Argument of binary operator missing	The first or second argument of a binary operator is missing.
80	Statement expected	One of ? : ,] or } was found where statement was expected.

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 uni on 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)	... cannotbe 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 i nt 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.
99	Extended keyword not allowed in this context	An extended processor-specific keyword occurred in an illegal context; eg `interrupt int i`.

No	*Error message*	*Suggestion*
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 si zeof operator was applied to abit-field, function, or extern array of unknown size.

104	Initializer "string" exceeds array dimension
	
	For example:
	
	`char array [ 4 ] = "abode";`
	`/* invalid */`
		An array of char with explicit dimension was initialized with a string exceeding array size.

105	Language feature not i mpl emented: '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 requi red	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 (*(*pl)[8])(int);
char (*(*p2)[8])(float);

/* pi 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 (*pl)[8];
int (*p2)[9];

/* pi 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, #i f, #el i f, 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 - poi nter. 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 #e r r o r 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 #i f and #el i f 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 maybe 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(pi)      #pl              /* Becomes "pi" */
#define mac(pl,p2)   pl+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 enumerati on tag definition that has no objects declared at the same time. Also, functions are considered as being unable to return const or vol ati 1 e.

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 (possiblyindirect) 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&Rfunction 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

D

E

F

G

H

I

IV

L

M

N

O

p

Q

R

S

T

u

V

W

X