User’s Guide

HP B3640 Motorola 68000 Family C Cross Compiler

Notice

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

MS-DOS and Windows are U.S. registered trademarks of Microsoft Corporation.

Hewlett-Packard Company
P.O . Box 2197
1900 Garden of the Gods Road
Colorado Springs, CO 80901-2197, U.S.A.

RESTRICTED RIGHTS LEGEND Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in subparagraph (C) (1) (ii) of the Rights in Technical Data and Computer Software Clause in DFARS 252.227-7013. Hewlett-Packard Company, 3000 Hanover Street, Palo Alto, CA 94304 U.S.A.

Rights for non-DOD U.S. Government Departments and Agencies are set forth in FAR 52.227-19(c)(1,2).

About this edition

Many product updates and fixes do not require manual changes, and manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual revisions.

Edition dates and the corresponding HP manual part numbers are as follows:

Edition 1	B3640-97000, May 1993
Edition 2	**B3640-97001, January 1994**

B3640-97000 incorporates information which previously appeared in 64902-92003, 64902-97000, 64902-97001, 64903-92004, 64903-97000, 64903-97001, 64907-92002, 64907-97000, 64907-97001, 64908-92002, 64908-97000, 64908-97001, 64909-92002, 64909-97000, and 64909-97001.

Certification and Warranty

Certification and warranty information can be found at the end of this manual on the pages before the back cover.

Features

The Motorola 68000 Family C Cross Compiler translates C source code into 68000 family assembly language which can be accepted by the HP B3641 assembler. This compiler has special features to help meet the needs of the embedded system designer:

- ANSI standard C compiler and preprocessor.
- Standard command line interface for compatibility with **make** and other utilities.
- Complete C support and math libraries from ANSI standard for nonhosted environments.
- In-line code generation and libraries to support the 68881/2 floating point unit.
- Three ways to embed assembly language in C source.
- Named section specification in C source.
- Choice of address modes for function calls and static data access.
- Option to copy initial value data from ROM to RAM at load time.
- Listings with generated assembly language, C source, and cross references.
- Fully reentrant generated code.
- Optimization for either time or space.
- Constant folding, automatic register variable selection, and other global optimizations.
- Full symbol information and C source line numbers provided for debugging, emulation, simulation, and analysis tools.
- Compiler reliability ensured through object-oriented design and exhaustive testing.

Contents

Part 1 Quick Start Guide

1 Getting Started

Part 2 Compiler Reference

8 Environment-Dependent Routines

12 Behavior of Math Library Functions

13 Comparison to C/64000

Part 1

Quick Start Guide

Part 1

1

Getting Started

How to get started using the compiler.

In this chapter

This chapter contains the following information:

- An overview of the C compiler.
- Instructions for common tasks, such as compiling a simple program.
- Short examples so you can practice the common tasks.

What you need to know

Before you begin to learn how to use this compiler, you should be familiar with the following:

- The C programming language.
- The Motorola 68000 family microprocessor architecture.
- Basic host operating system commands (such as **cp**, **mv**, **ls**, **mkdir**, **rm**, and **cd** in UNIX or **copy**, **dir**, **mkdir**, **del**, and **cd** in DOS) and a text editor (such as **vi** in UNIX or **edit** in DOS).

In addition, most sections in this manual assume that you are familiar with 68000 family assembly language.

Parts of the compiler

The "compiler" is really a set of programs:

- **cc68k**, the C compilation control command.
- **cpp68k**, the C preprocessor.
- **clst68k**, the lister.
- **ccom68***xxx*, the C compiler. (**ccm68***xxx* for DOS.)
- **opt68***xxx*, the peephole optimizer.

The compiler makes use of several assembler programs:

- **as68k**, the assembler.
- **ld68k**, the linking loader.

To compile a C program, you can use just the **cc68k** C compilation control command. The **cc68k** command will run the other programs as needed.

Summary of compiler options

-b Invoke Basis Branch Analyzer preprocessor. (UNIX only)

-B Cause generation of big switch tables for > 32K byte switch bodies.

-c Do not link programs (object files are generated).

-C Do not strip C-style comments in preprocessor.

-d Separate data into initialized and uninitialized sections.

-D *name[=def]* Define *name* to the preprocessor.

-e Fast error checking (no code is generated).

-E Preprocess only (send result to standard output).

-f Generate code to use the 68881/2 coprocessor.

-g Generate run-time error checking code (overrides **-O**).

-h Generate HP 64000 format (**.X**) files.

-I *dir* Change include file search algorithm.

-k *linkcomfile* Link using the *linkcomfile* linker command file.

-K Enforce strict section consistency.

-l*x* Search **lib***x***.a** (**lib***x***.lib** for DOS) when linking.

-L[**i**][**x**] Generate ".O" (".lst" for DOS) listing(s). The **-i** option causes include files to be expanded and included in the listing. The **-x** option causes cross-reference tables to be included in the listings. (Overridden by **-e**, **-E**, and **-P**.)

-m Specify addressing mode.

-M Cause generation of more warning messages than are generated by default.

-N Cause linking with **linkcom.k** (no I/O) rather than **iolinkcom.k** (**iolinkco.k** for DOS).

-o *outfile* Name absolute file *outfile* instead of **a.out.x**. (**aout.abs** for DOS).

-O[**G**][**T**] Optimize. **-O** for space, **-OT** for time, **-OG** for debugging.

-p *processor* Compile code for the specified processor.

-P Preprocess only (send result to **.i** files).

-Q Word align data in memory instead of default quad (double word) alignment.

-r *dir* Use default linker command file in /usr/hp64000/env/*dir* (\hpcc68k\env\dir for DOS) instead of the default.

-s Strip symbol table information (overridden by **-g** and **-L**).

-S Only generate assembly source files (with **.s** extensions).

-t *c,name* Insert subprocess *c* whose full path is *name*.

-u Consider non-constant static data uninitialized.

-U *name* Undefine *name* to the preprocessor.

-v Verbose (produce step-by-step description on *stderr*).

-w Suppress warning messages.

-W *c,args* Pass *args* as parameters to subprocess *c*.

Summary of file extensions

UNIX Extension	DOS Extension	Meaning	Where generated
.a	**.lib**	Archive (library) file	ar68k
.A	**.a**	HP 64000 format assembler symbol file	as68k
.c	**.c**	C source file	editor
.EA, .EB	**.ea, .eb**	Emulator configuration file	emulator interface or editor
.h	**.h**	Include (header) file	provided or editor
.i	**.i**	"Preprocess only" output	cc68k -P
.k	**.k**	Linker command file (default extension used by cc68k)	editor
.L	**.l**	HP 64000 format linker symbol file	ld68k -h
.1	**.txt**	On-line manual page	provided
.o	**.obj**	HP-MRI IEEE-695 format relocatable object file	as68k
.O	**.lst**	Listing file	cc68k -L
.s	**.s**	Assembly language source file	cc68k or editor
.x	**.abs**	HP-MRI IEEE-695 or Motorola S-Record absolute object file (executable)	ld68k
.X	**.x**	HP 64000 format absolute file (executable)	ld68k -h (via cc68k -h)
.Ys	**—**	Symbol file directory	emulator interface

To install the software on a UNIX workstation

1 Load the software from the software media.

Instructions for installing the software are provided with the software media, or in your operating system’s system administration guide.

2 Set the HP64000 environment variable.

Set this variable to the location of the software, usually /usr/hp64000.

3 Set the MANPATH environment variable.

Add $HP64000/man to this variable so that you can read the on-line "man pages."

4 Set the PATH environment variable.

Add $HP64000/bin to your path so that you can run the compiler programs.

You should add these commands to your .login, .vueprofile, or .profile file (if they are not there already) so that you won’t need to re-enter them every time you log in.

Examples

If you installed the compiler in the root directory on an HP-UX system, enter:

```
export HP64000=/usr/hp64000
export PATH=$PATH:$HP64000/bin
export MANPATH=$MANPATH:$HP64000/man
```

On a Sun system, you would enter:

```
setenv HP64000 /usr/hp64000
setenv PATH $PATH:$HP64000/bin
setenv MANPATH $MANPATH:$HP64000/man
```

To install the software on a PC (Windows)

To install from MS Windows:

1 Start MS Windows in the 386 enhanced mode.

2 Insert compiler Disk 1 into floppy disk drive A or B.

3 Choose the File→Run... (ALT, F, R) command in the Windows Program Manager. Enter "a:\setup" (or "b:\setup" if you inserted the floppy disk into drive B) in the Command Line text box.

4 Choose the OK button. Follow the instructions on the screen.

You will be asked to enter the installation path. The default installation path is c:\hpcc68k. The default installation path is shown wherever files are discussed in this manual.

5 Edit your AUTOEXEC.BAT file.

The Setup program will create a file called AUTOEXEC.AXL which shows how to set the PATH, HPCC68K, and HPAS68K variables in your AUTOEXEC.BAT file. If you have multiple configurations or your AUTOEXEC.BAT file starts a shell or Windows, be careful to place the SET and PATH commands at the appropriate place in the file.

When you have edited the AUTOEXEC.BAT file, you need to reboot your computer to set these environment variables.

Notes

To follow the examples in this chapter, you need to get to a DOS prompt. You can do this by leaving Windows or by opening an MS-DOS window.

You must use the Windows Setup program to install the compiler.

Unless otherwise noted, the example listings, file names, and paths in this manual are for HP-UX systems. Use c:\hpcc68k in place of /usr/hp64000 or $HP64000. DOS file extensions are listed on page .

System requirements

The compiler requires the following configuration:

- An IBM Personal Computer, HP Vectra, or 100 percent compatible
- MS-DOS version 3.3 or later
- MS Windows version 3.0 or later
- An 80386 processor or higher
- 4 Mbytes of available memory (RAM)
- Hard disk with at least 4 Mbytes of free space. At least 11 Mbytes is recommended.
- A 1.2 Mbyte, 5.25-inch floppy disk drive or a 1.44 Mbyte, 3.5-inch floppy disk drive

To remove unnecessary files (UNIX only)

- If the compiler is using too much disk space, you can remove files for any processors you will not be using.

  You may remove the following files from the $HP64000 directory:

  - lib/ccom*processor*
  - lib/opt*processor*
  - lib/*processor*/*
  - include/*processor*/*

  where *processor* is any processor for which you will not need to compile any code.

To create a simple C program

- Use a text editor to create the file simple.c:

```
#include <stdio.h>

main()
{
        char    *str = "a string";

        printf("\nThe string is: \"%s\"\n", str);
}
```

Figure 1-1. The "simple.c" Example Program

To compile a simple program

- Use the cc68k comand at your host operating system prompt.

Example

To compile the "simple.c" example program, enter the following command:

```
cc68k simple.c
```

This command generates the executable file **a.out.x** (or **aout.abs** for DOS) by default. The compiler will generate the code for the 68000 by default.

The UNIX version of the compiler will print a warning message because a target processor was not specified. Because this is just an example, ignore the warning.

To generate an assembly listing

- Use the **-L** compiler option.

This option generates a listing of the C source, which includes the generated assembly code, and a linker listing.

Example

To generate the listings for "simple.c", enter:

```
cc68k -L simple.c
```

The mixed source and assembly listing is sent to file **simple.O**, and a linker listing is sent to file **a.out.O**. (These files are named **simple.lst** and **aout.lst** if you are using the compiler on a DOS system.)

Examine the **simple.O** file and note how:

- Addresses of strings are passed as parameters to the "_printf" support library routine (String1+0 is pushed, then _printf is called).
- String literals are placed in the "const" section.

Now look at **a.out.O** and note that:

- The file shows the default linker command (generated by the compilation control command).
- The linker command is followed by the contents of the default linker command file. The default linker command file loads some libraries and an emulation monitor or monitor stub.
- Modules are listed in the order they are loaded. Modules within library files are listed in alphabetical order.
- The module crt0 is the program setup routine. Program execution will begin with this routine.

To specify addressing modes

- Use the **-m** command line option to specify the addressing mode for a section.

The 68000 C Compiler allows you to select the addressing modes used in the generated assembly code for accessing data and calling functions (branches are always done PC relative).

By default, the absolute long (the most flexible) addressing mode is used. Addressing modes are selected using named sections (which are also used in the linker when specifying load addresses).

To name sections in the source file, use the **SECTION** pragma.

Example

To specify that program code and constants (the ROMable portion) be placed in section *MyProg* and data be placed in section *MyData*, insert the line

```
#pragma SECTION PROG=MyProg DATA=MyData CONST=MyProg
```

at the beginning of the C source. These section names now apply not only to code and data generated in the source file, but also to any extern functions or data referenced in the file.

To specify that absolute short addressing be used from section *MyProg* to section *MyData*, use "**-**m MyProg,MyData,as" ("as" is an abbreviation of absolute short). This would be appropriate if *MyData* is located in the base page.

A simple example

To add two static integers and place the result in a third integer which is an extern:

```
#pragma SECTION PROG=MyProg DATA=MyData CONST=MyProg
extern int a;
int b,c;
main(){
  a=b+c;
}
```

You can compile this example using the following command:

```
cc68k  -LOc  -m MyProg,MyData,as  small.c
```

The small.O (small.lst for DOS) listing file looks like this:

```
HPB3640-19300  68000 C Cross Compiler A.04.00  small.c

             HPB3641-19300 A.02.00 19Apr93 Copr. HP 1988 Page 1 Mon Apr 26 15:09:55 1993


Command line: as68k -Lfnot,llen=1100 -o small.o /tmp/ct3CAAa27665
Line Address
                                        CHIP    68000
                                        NAME    small
                                *
                                * MKT:@(#) B3640-19300 A.04.00 MOTOROLA 68000 FAMILY C
                                  CROSS COMPILER
                                *
                                * Assembler options:
                                *
                                        OPT     BRW,FRL,NOI,NOW
                                *
                                * Macro definition for calling run-time libraries:
                                * bytes per call = 6
                                *
                                CALL    MACRO   routine
                                          XREF  routine
                                          JSR   (routine).L
                                        ENDM
                                *
                                        SECT    MyProg,2,C,P
    1   #pragma SECTION PROG=MyProg DATA=MyData CONST=MyProg
    2   extern int a;
    3   int b,c;
    4   main(){
                                        XDEF    _main
                                _main
    5     a=b+c;
     00000000 2038 0000       R         MOVE.L  (_b+0).W,D0
     00000004 D0B8 0004       R         ADD.L   (_c+0).W,D0
     00000008 21C0 0000       E         MOVE.L  D0,(_a+0).W
    6   }
                                functionExit1
                                returnLabel1
     0000000C 4E75                      RTS
                                        XREF    MyData:_a
                                        SECT    MyData,2,D,D
                                        XDEF    _b
                                        ALIGN   2
                                _b
     00000000 ==00000004=of=            DCB.B   4,0
              00
                                        XDEF    _c
                                        ALIGN   2
                                _c
     00000004 ==00000004=of=            DCB.B   4,0
              00
                                        END
```

Note that variables "a", "b", and "c" are accessed using the absolute short addressing mode (indicated by the .W extension).

Next, change the example slightly to put variable "a" in another section (the default is *data*)

```
extern int a;
#pragma SECTION PROG=MyProg DATA=MyData CONST=MyProg
int b,c;
main(){
  a=b+c;
}
```

Compile the same way as before. Variable "a" will be declared external in section data and referenced absolute long rather than absolute short (as indicated by the .L extension).

An example using A5 relative addressing

A5 relative addressing allows accessing data values as offsets from an address loaded into the A5 address register at program startup. The most common use of this addressing mode is to create a second "basepage" (that is, a 64K byte block of memory that can be accessed more efficiently than by absolute addressing). For example, here is a short program that accesses variable "x" using absolute long addressing, variable "i" using absolute short addressing, and variable "q" using A5 relative addressing:

```
int x;
#pragma SECTION DATA=0x400 BP
int i;
#pragma SECTION DATA=SecondBasePage
int q;
main(){
  x=q+i;
}
```

You can compile this program using the following command:

```
cc68k -LOc -m all,SecondBasePage,a5s short.c
```

The SECTION pragma used to locate variable "i" specifies an absolute address where "i" is "ORG’d" and the appended "BP" tells the compiler that the address is on basepage (and, thus, to use absolute short addressing). The **-m** option specifies that **all** references to the section named SecondBasePage be done A5 relative short. In addition to the addressing mode selection, for A5 relative addressing, the linker must be told the run-time value of A5. This is accomplished via an INDEX command in the linker command file such as:

```
INDEX ?A5,SecondBasePage+$8000
```

The setup routine (crt0) initializes A5 to the value assigned by the linker to special symbol ?A5. In this case, A5 will be initialized to the address 8000(hex) above the start of the section named SecondBasePage. This allows up to 64K bytes of data in section SecondBasePage to be accessed A5 relative short.

An example using PC relative addressing

The PC relative addressing mode is most commonly used for branches (which are always done PC relative on the 68000), function calls, and reads of constant data. The 68000 does not support PC relative writes, but the compiler does synthesize this "addressing mode" using multiple instructions. When a group of mutually referencing functions fits into 64K bytes, it is more efficient to use PC relative short calls between them.

If you would like to see an example, create this program:

```
int f(int i);

main() {
  int i,x;
  i=f(x);
}

int f(int i) {
  return i*2;
}
```

Then compile the program:

```
cc68k -LOc -m prog,prog,pcs pcs.c
```

Here, function "f" is called PC relative short (rather than the default "JSR (_f+0).L"). By combining PC relative and A5 relative addressing modes, one can create a variety of position independent code modules.

An example using run-time and support libraries

Run-time library routines are called *implicitly* by the generated assembly code (for example, dtoi is called to cast a double to an int). Since these implicitly called routines are not visible in the C source, a special section named **lib** is reserved and understood by the compiler to be the section in which run-time libraries are defined. Furthermore, a restriction is placed on addressing modes used to call run-time libraries: all calls to run-time libraries from a single C source file must use the same addressing mode. For this reason, **all** is the only section allowed in a **-m** option specifying **lib** as its destination (use -m all,lib,mode).

Support library routines, unlike run-time library routines, are called *explicitly* in the C source. Thus, they behave just as though they were user-written functions from an addressing mode specification point of view. Their section names are the same as the base name of the library (for example, **libc.a**'s section is **libc**).

On the PC host, there is only one library (**lib** and **libc** come from the same library file).

For example, assume that the run-time library **lib.a** (named section **lib**) is located on base page and that the support library **libc.a** (named section **libc**) is loaded in the same 32K byte memory block as the following program:

To call run-time library routines using the absolute short addressing mode and support library routines using the PC relative short addressing mode, you might compile the program using the command line

```
cc68k  -LOc  -m all,libc,pcs  -m all,lib,as  libcalls.c
```

Note that it is important to use "#include <stdio.h>" since without it the compiler does not know that "printf" is in named section **libc**.

To specify the target microprocessor

- Name the target microprocessor on the command line using the **-p** option.

You can specify the following target processors:

- 68000
- 68EC000
- 68HC000
- 68HC001
- 68010
- 68302
- 68020
- 68EC020
- 68030
- 68EC030
- 68040
- 68EC040
- 68331
- 68332
- 68340
- 68360
- CPU32

Example

To compile the example program for the 68020, enter

```
cc68k -p 68020 simple.c
```

To always compile for the 68020, enter:

```
export CC68KOPTS="-p 68020"
```

This is the same as entering "-p 68020" on the command line every time you use cc68k. Use **setenv** instead of **export** on Sun systems. Use **set** instead of **export** on DOS systems.

To compile for a debugger

To gain the most benefit from HP debuggers and emulators, follow these guidelines:

- Use the **-OG** option to generate debugging information.
- Avoid optimizing modes (**-O** or **-OT**).
- Turn off the automatic creation of register variables (**-Wc,-F**).
- Do not use the **-h** option. HP debuggers now use **.x** (**.abs** for DOS) rather than **.X** files.
- Use the C compiler's floating point library routines to generate code that will run interchangeably in both the debugger/simulator and the debugger/emulator.
- Use the same environment files as you would use to compile for an HP 64700-series emulator.

Example

To compile the simple.c program to be run in a debugger, use the following command:

```
cc68k -LM -OG simple.c
```

See Also

See the *User's Guide* for your debugger/emulator, debugger/simulator, or emulator interface for information on how to run a program in the debugger or emulator environment.

To use a makefile (UNIX systems only)

The UNIX **make** command can simplify the process of compiling your programs. This command allows you to specify which files are dependent on which other files (for example, **make** "knows" that files which end in **.o** are produced by compiling corresponding files that end in **.c** or by assembling programs that end in **.s**). If your host operating system is HP-UX, see the man page for **make** in section 1 of the *HP-UX Reference Manual*. See also "Make, a Program for Maintaining Computer Programs" in the "Programming Environment" volume of *HP-UX Concepts and Tutorials*.

Because **cc68k** is similar to the host **cc** command, it is easy to tell **make** how to compile, assemble, and link using cross tools. To any makefile designed for the host, you need to add some definitions and set up some options. These are:

```
CC=/usr/hp64000/bin/cc68k
AS=/usr/hp64000/bin/as68k
LD=/usr/hp64000/bin/ld68k
```

These definitions will cause **make**’s "built-in" rules to access the cross tools, and because the built-in options mean the same thing to the cross tools as they do to the host tools, the built-in rules now work when invoking the cross tools.

Note

The SunOS **make** command adds a "-target" option to the compiler command line. To remove this option, add the following statement to the beginning of the makefile:

```
COMPILE.c= $(CC) $(CFLAGS) $(CPPFLAGS) -c
```

Make also has a mechanism for passing additional options to the compiler, assembler, and linker. The additional options are passed each time the program is invoked and are thus set only for "global" options. For example, to always have the compiler and assembler produce listings, one might use:

```
CFLAGS = "-L"
ASFLAGS = "-Lfnot"
```

Some versions of **make** give default values for these options.

Here is an example makefile:

```
# These definitions are added to use the cc68k cross tools.

CC = cc68k
```

```
# All object files (make knows how to generate them from
# sources based on implicit rules).

OBJECTS = main.o file1.o grammar.o

# This dependency links the program together.

program.x: $(OBJECTS)
           $(CC) $(OBJECTS) -o program.x

# This dependency causes make to recompile file1.c
# whenever file1.h has been touched.

file1.o: file1.h
```

When run in a directory containing sources:

```
main.c    file1.c   grammar.y    file1.h
```

The commands generated by HP-UX **make** will be:

```
cc68k -O -c main.c
cc68k -O -c file1.c
yacc grammar.y
cc68k -O -c y.tab.c
rm y.tab.c
mv y.tab.o grammar.o
cc68k main.o file1.o grammar.o -o program.x
```

This example assumes that **/usr/hp64000/bin** has been added to your PATH environment variable.

You can see what commands will be generated by **make** by using the following command:

```
make -n
```

To modify environment libraries

To modify the environment-dependent library **env.a** (**env.lib** for DOS), the startup routines **crt0.o** or **crt1.o** (**crt0.obj** or **crt1.obj for DOS), or the monitor stub mon_stub.o** (**mon_stub.obj** for DOS):

1 Copy the source files.

The following command copies the environment-dependent source files to the current directory. The "**.**" just before the return means that the names of the files are not changed.

```
cp /usr/hp64000/env/hp<emulator_environment>/src/*  .
```

Or, on a DOS system, enter:

```
copy c:\hpcc68k\env\hp<emulator_environment>\src\* .
```

2 Edit the source files.

The following command changes the permissions of the source files so that you will be able to save any changes you make while editing the files.

```
chmod  644  *
```

Or, on a DOS system, enter:

```
attrib -r *.*
```

Now you may edit the source files as needed.

3 Set up the directory structure for "Makefile". (UNIX only)

The **make** utility will be used to create a new environment dependent library which contains the changes made to the source files. As provided, **Makefile** assumes there are two directories under the directory in which it resides, "src" and "obj". **Makefile** also assumes that all source files are in the "src" directory. The following commands set this situation up.

```
mkdir  src
mkdir  obj
mv  *.s *.c src
```

4 Run the "make" command. (UNIX only)

The following command will create a new environment-dependent library file **env.a**, new startup modules **crt0.o** and **crt1.o**, and the monitor stub **mon_stub.o**, and will place them in the current directory.

```
make  all
```

In addition to the **all** target, other targets are available for the **make** command which will create only those files needed. A list of these available targets is displayed by the following command.

```
make  help
```

The following command will remove unnecessary intermediate files left by the **make all** command.

```
make  clean
```

5 Compile all of the source files. (DOS only)

For example, to compile for the 68020, enter:

```
\hpcc68k\cc68k -p 68020 -Ou -Wc,-i -c *.c
\hpas68k\as68k -fnod -fp=68020 *.s
```

6 Create the **env.lib** library. (DOS only)

Create a temporary file "ar_cmd", similar to the following:

```
CREATE env.lib
ADDMOD disp_msg.obj
ADDMOD trap.obj
ADDMOD getmem.obj
ADDMOD heap.obj
ADDMOD stack.obj
ADDMOD startup.obj
ADDMOD open_file.obj
ADDMOD systemio.obj
ADDMOD sbrk.obj
ADDMOD fpu_trap.obj
```

```
SAVE
END
```

Add **fpu_trap.obj** only for the 68020, 68030, or 68040.

Next, use **ar68k** to make the library file:

```
\hpas68k\ar68k < ar_cmd
```

7 Modify the default linker command file.

The following UNIX commands copy the default I/O linker command file to the current directory so that you can edit it to load the environment file just created. (Copy **linkcom.k** if your programs do not use I/O.)

```
cp  /usr/hp64000/env/hp<emulator_environment>/iolinkcom.k  .
chmod  644  iolinkcom.k
vi  iolinkcom.k
```

Change the line which reads

```
LOAD  /usr/hp64000/env/hp<emulator_environment>/env.a
```

to

```
LOAD  env.a
```

The equivalent DOS commands are:

```
copy c:\hpcc68k\env\hp<emulator_environment>\iolinkco.k
attrib -r iolinkco.k
edit iolinkco.k

LOAD env.lib
```

Similarly, if you have modified the startup module source file **crt0.s** or **crt1.s**, or the monitor stub **mon_stub.s**, you should also change the linker command file so that it loads the local version instead of the shipped version.

Specifying the modified linker command file when compiling your program (with the **-k** option) will cause the linker to call in routines from the modified environment-dependent library.

About environment libraries

Many files are linked into the C program from the environment libraries. These libraries reside in the subdirectories of **/usr/hp64000/env** (**\hpcc68k\env** for DOS) and are designed to support the emulator (and simulator, if available). But these do more than just help you use the emulator.

The C compiler has only limited information about the environment in which compiled programs will ultimately execute. All the high level functions depend on the environment libraries to provide the low level hooks into the execution environment (or target system). The supplied environment libraries provide the hooks necessary to operate in the emulator environment. They also serve as a pattern for you to create your own low level hooks to allow the C compiler to work in your own execution environment. You may either modify our environment files (the source code is provided) or use the files as a pattern to create your own equivalent files. HP has made every effort to narrow this "hook-up point" as much as possible, but you will need to make some modifications in order to run your programs in your own execution environment.

To view the on-line man (help) pages

- On a UNIX system, use the **man** command.

- On a DOS system, use the **more** command or an editor.

You can display on-line "man pages" for any of the programs which make up the Motorola 68000 Family C Cross Compiler:

- **cc68k**
- **cpp68k**
- **clst68k**

Refer to the on-line man pages for detailed information about command-line options and compiler directives.

Because the man pages contain important information which is not included in this manual, HP recommends that you print the *cc68k* man page and keep it near your computer.

On UNIX systems, the man pages are in the directory $HP64000/man. If the **man** command cannot find the man pages, check that you have added this directory to the MANPATH environment variable.

On DOS systems, the help files are in the directory \hpcc68k.

Example

(UNIX) To view the **cc68k** on-line manual page, type the following command from the operating system prompt:

```
man  cc68k
```

(DOS) To view the **cc68k** help file, type the following command at the DOS prompt:

```
more < c:\hpcc68k\cc68k.txt
```

Information on the **cc68k** compiler syntax and options will be scrolled onto your display.

Part 2

Compiler Reference

Part 2

2

C Compilation Overview

An overview of the Motorola 68000 Family C Cross Compiler and a description of the ANSI C language.

Execution Environment Dependencies

Providing the "standard I/O" and storage allocation C library functions creates dependencies on the environment in which programs execute.

Since the C compiler is a tool to help you develop software for your own target system execution environments, HP has been careful about any execution environment dependencies associated with this compiler or its libraries.

The compiler provides the "standard I/O" and storage allocation library functions; therefore, there are some environment dependencies to be aware of. The compiler isolates these environment dependencies to make it easier to tailor the compiler to your own target system execution environment.

The execution environment-dependent routines provided with the C compiler are written to work in the HP development environments, but they need to be rewritten for target system execution environments.

C Compilation Overview

An overview of the C compiler is shown in figure 2-1. The entire process is controlled by the command line fed to the compilation control routine. Rectangles in the diagram represent either data provided by the programmer (C source file, for example) or data produced by one of the circular processes (output listing, for example). Each process is described following the figure.

In the following figure, the names of programs appear in parentheses. These names refer to the cross tools, and not to the native tools. For example, "cc" refers to **cc68k** cross compiler and not to the native host **cc** compiler.

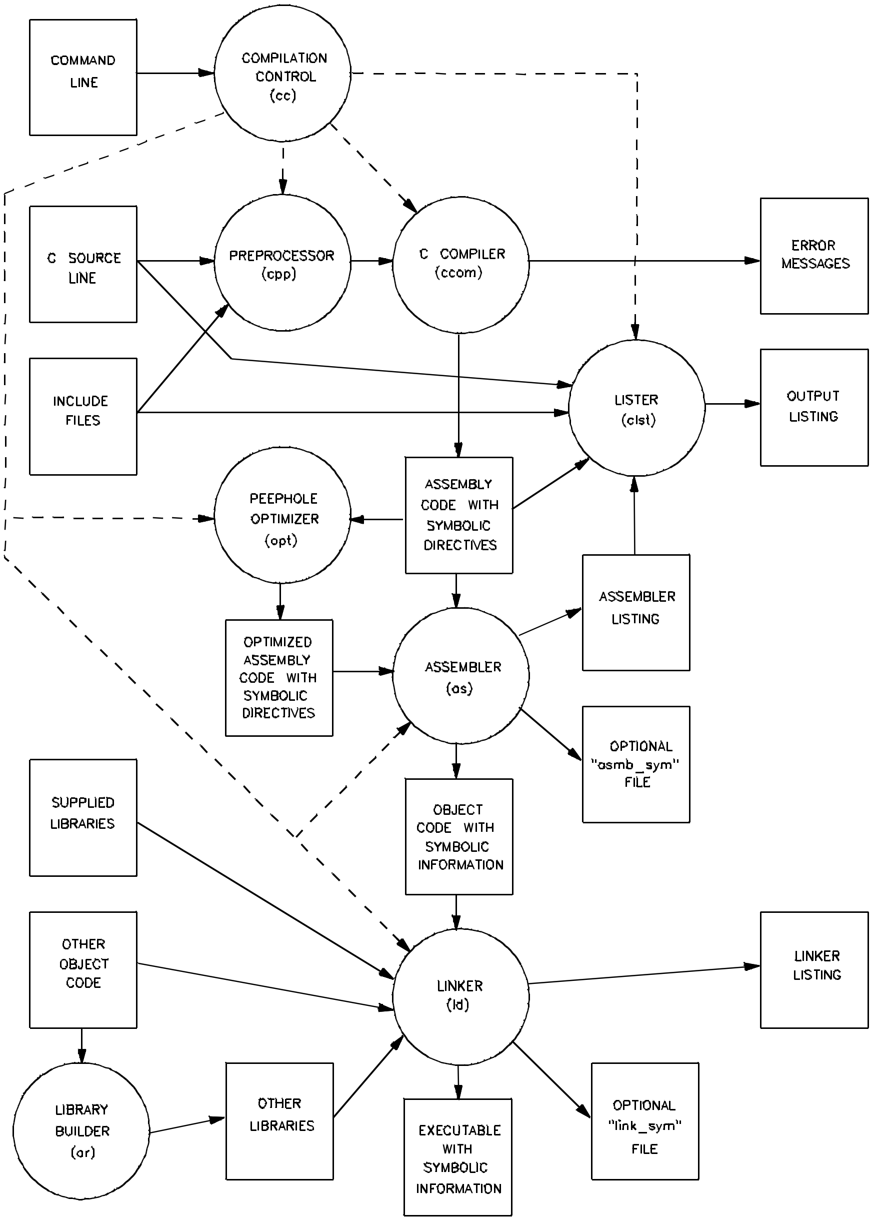


Figure 2-1. C Compilation Overview

Compilation Control Routine

The entire system is controlled by a compilation control routine, cc68k. The compilation control routine calls in sequence: the C preprocessor (cpp68k), the C compiler (ccom68*xxx* on UNIX systems or ccm68*xxx* on DOS systems), optionally the peephole optimizer (opt68*xxx*), the assembler (as68k), optionally the lister (clst68k), and the linker (ld68k). Many of these programs may be run individually using the cc68k command's options. See the on-line man pages for the description of the command syntax and options.

The librarian (ar68k) is a separate tool for building archive files used by the linker.

C Preprocessor

The 68000 family C preprocessor accepts C preprocessor directives which modify the source code that the compiler sees. This modification includes expansion of include files, expansion of macros, and management of conditional compilation. See the on-line man page for a description of the C preprocessor.

C Compiler

The C compiler accepts C language as defined by the ANSI C Standard. The compiler performs a translation with optional optimizations (see the "Optimizations" chapter) and emits an assembly language source file containing embedded directives which provide information to be used by the lister and later by the debugger and analyzer (see the "Compiler Generated Assembly Code" chapter). The compiler also emits error and warning messages to the standard error output. These messages include the original source line on which the error occurred with a pointer to the offending token.

Peephole Optimizer

The peephole optimizer is run when the "optimize" command line option is specified. It performs peephole optimization on the assembly output of the compiler. The optimizer makes allowances for **volatile** data types and embedded assembly code to avoid changing the functionality of the generated code. The optimizer works properly only on compiler-generated assembly code and is not a stand alone tool for use on hand-written assembly code. Refer to the "Optimizations" chapter for more information on the peephole optimizer.

Assembler

The assembler is the HP B3641 assembler which accepts an assembly language source file (optionally containing symbolic debug information defined by special directives) and produces an object code file (optionally containing a representation of the symbolic debug information from the assembly source) and an optional listing for use by the lister in generating the final listing. The assembler also has a switch for generating HP 64000 format assembler symbol files.

Source File Lister

The source file lister is run when the "listing" command line option is specified. The lister uses the assembler source or listing, C source file, and include files to produce a listing. The listing includes embedded assembly language and, optionally, expanded include files and a cross reference table. The lister is controlled by "*LINE*" directives inserted by the compiler into the output assembly code. Because the lister is usually run by the compilation control routine, details of the lister directives are not described in this manual. See the on-line man page for the description of clst68k command syntax and options.

Librarian

The librarian is the HP B3641 librarian which combines several object code files (generated by the assembler) into an archive file which the linker will search when it tries to resolve external references. The libraries that are part of the compiler product are made with this librarian.

Linker

The linker is the HP B3641 linker which accepts several object code or archive files (generated by the assembler or librarian, respectively) and creates an absolute file containing all object code and symbols to be loaded. Optional load maps may be generated as well as HP 64000 format linker symbol and absolute files.

ANSI Extensions to C

The B3640 Motorola 68000 Family C Cross Compiler complies with ANSI/ISO standard 9899-1990. In some cases, programs which compile with no errors on old C compilers will result in errors or warnings with this compiler. Although this may seem inconvenient, modifying the source will result in portability to other ANSI standard C compilers.

Assignment Compatibility

The ANSI standard has more carefully regulated assignment compatibility. In particular, pointers and integers are no longer considered to be assignment compatible without casts, and pointers to different typed objects are not assignment compatible without casts.

Pointers and Integers

Because assignments between pointers and integers occur often in many existing C programs, such assignments are warned rather than being flagged as errors by the Motorola 68000 Family C Cross Compiler. It is still recommended practice not to perform such assignments without casts.

Pointers and Pointers

The assignment of a "pointer to one type" to a "pointer to another type" only generates a warning message. However, the ANSI standard has provided a new type (**void**) to which a pointer may point; the resulting "pointer to void" may be assigned to any pointer.

Function Prototypes

Function prototypes allow you to specify the types of function parameters and whether a function accepts variable parameters. They allow the compiler to check the consistency of parameter types between declarations and calls of a function in a file. Because the linker does not check for incompatible calls across file boundaries, we recommend that you use an include file to declare the function at all reference and definition points.

Function prototype information is used by the compiler to generate more efficient code by *not* widening passed parameters. That is, **short** and **char** passed parameters are not widened to **int**; and **float** parameters are not widened to **double**, as is the case in the absence of function prototypes.

Old style function declarations (those without any parameter information) continue to have the same meaning as before. All **short** and **char** parameters are widened to **int**, and all **float** parameters are widened to **double** at the function call. The appropriate inverse conversions are performed at function entry. Old style and prototype declarations for the same symbol can coexist as long as all of the parameter types specified in the prototype are the widened types and as long as the ellipsis is not used. It is good practice to convert all declarations to prototype syntax if prototypes are going to be used.

The consistency checking between the type of expression passed as a parameter to a prototyped function and the declared type of the corresponding parameter requires that the two types be assignment compatible. The parameter expression will be converted to the formal parameter type prior to its value being passed.

The following is an example of function prototype usage:

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

/* Note the optional use of identifier "format" to document the parameter’s
   meaning.  The ellipsis indicates zero or more additional parameters. */

extern float float_operation(float,float);

/* In this case, only type names are given for the parameters. */

/* The following is the prototype syntax for a function definition. */

void func(int i)
{
        float f;

        f = float_operation(i, 2.0);

        /* The int "i" and the double "2.0" will be converted to float
           before being passed (the "2.0" is converted at compile time).
           Both parameters are passed as floats without the expensive
           run time conversion to double which old style functions cause. */
}
```

Pragmas

Pragmas are special preprocessor directives which allow compilers to implement special features. By definition, any pragma that a compiler does not understand will be ignored. However, because pragmas allow compilers to deviate from the standard, their number has been kept to a minimum.

The pragmas which the C compiler understands are listed below. Pragmas which are not recognized cause a warning message to be written to the standard error output.

#pragma SECTION

Provides for renaming the default program section names. (Refer to the "Section Names" section of the "Embedded Systems Considerations" chapter for more information.)

#pragma ASM/END_ASM

Provides for including assembly language in the C source file. (Refer to the "Using Assembly Language in the C Source File" section of the "Compiler Generated Assembly Code" chapter for more information.)

#pragma FUNCTION_ENTRY/EXIT/RETURN "C_string"

Provides for including assembly language instructions in the function entry and exit code of the compiler-generated assembly code. (Refer to the "Using Assembly Language in the C Source File" section of the "Compiler Generated Assembly Code" chapter for more information.)

#pragma INTERRUPT

Provides for implementing functions as interrupt routines. (Refer to the "Implementing Functions as Interrupt Routines" section of the "Embedded Systems Considerations" chapter for more information.)

#pragma ALIAS

Provides for the naming of an assembly language symbol associated with a C source file symbol. (Refer to the "Assembly Language Symbol Names" section of the "Compiler Generated Assembly Code" chapter for more information.)

The *void* Type

A new type, **void**, has been added by ANSI. It has two fundamental purposes. The first is to allow a function to be defined to have no return value (i.e., a procedure). Since **void** typed objects cannot be assigned to other objects, such procedures cannot be used in a context where a return value is required. (Of course, the protection afforded by this mechanism is limited to programs where functions are declared with a **void** return type using old style declarations or function prototypes.)

The second use of type **void** is to declare generic pointers. By definition, pointers to **void**, e.g., "void *genericPtr;", are assignment compatible with pointers to any other type. This can also be a convenient type for the return type of a function such as *malloc* whose result is then assignment compatible with any pointer.

The *volatile* Type Modifier

The type modifier **volatile** specifies that a particular variable's value may change from one read to another or following a write. An obvious example of such a "variable" is an I/O port in an embedded system. The **volatile** type modifier informs the compiler of this behavior so that the compiler can avoid performing optimizations which assume that variables' contents are not changed unexpectedly. (Refer to the "Effect of *volatile* Data on Peephole Optimizations" section in the "Optimizations" chapter; also, refer to "The *volatile* Type Modifier" section in the "Embedded Systems Considerations" chapter for examples of its use.)

The *const* Type Modifier

An object declared with the **const** type modifier tells the compiler that the object cannot be assigned to, incremented, or decremented; statements which attempt to do so will cause errors. Pointers to **const** storage cannot be assigned to pointers to non-**const** storage. Objects declared with the **const** type modifier can be accessed, but they cannot be written to. An object declared with the **const** type modifier, which has **static** storage class, is placed in the CONST section (see the "#pragma SECTION" section in the "Embedded Systems Considerations" chapter). Some examples of how the **const** type modifier is used follow.

```
static const char       message[][7] = {
                                "First ",
                                "Second",
                                "Third "
                        };

const char      *cnst_chr_ptr;  /* The pointer may be modified,    */
                                /* but that which it points to     */
                                /* may not.                        */

char *const     ptr;            /* The pointer may not be modified,*/
                                /* but that which it points to may.*/

const char *const      ptr;     /* Neither the pointer nor that    */
                                /* which it points to may be       */
                                /* modified.                       */
```

Translation Limits

The ANSI C Standard has set standard translation limits which must be met or exceeded by conforming implementations. The following list meets or exceeds all such limits put forth by the standard.

- Approximately 50 nesting levels in compound statements, iteration control structures, and selection control structures.
- Unlimited levels of nesting in preprocessor conditional compilation blocks.
- Approximately 100 pointer, array, and function declarators modifying a basic type in a declaration.
- Limited to 128 levels of expression nesting.
- There are 255 significant case-sensitive characters in an internal identifier.
- There are 255 significant case-sensitive characters in a macro name.
- There are 30 significant case-sensitive characters in an external identifier.
- Limited to 2^{31}-1 bytes of local variables in one function block.
- Unlimited simultaneous macro definitions.
- Limited to 2^{31}–1 bytes of parameters in function definition and call.
- Limited to 127 parameters in preprocessor macro.
- Limited to 1024 characters in a logical source line.
- 1023 characters in a single string literal (1024 including a trailing null character). There is no limit on the size of string made from adjacent string literals.
- Limited to 2^{31}-1 byte-sized objects.
- Unlimited nesting levels of include files.
- Unlimited number of cases in a switch statement.
- Size of the switch statement body is limited to 32767 bytes of generated code unless the "big switch tables" option to cc68k is specified (in which case, the size of the switch statement body is limited only by the size of the processor address space).

3

Internal Data Representation

How arithmetic and derived data types (arrays, pointers, structures, etc.) are represented in memory.

This chapter does not describe how to use data types in your programs. Refer to *The C Programming Language* for information such as escape sequences, **printf** conversions, and declaration syntax.

Arithmetic Data Types

The arithmetic data types are listed in the following table:

Table 3-1. Arithmetic Data Types

Type	# of Bits	Range of Values (Signed)	(Unsigned)
char	8	–128 to 127	0 to 255
short	16	–32768 to 32767	0 to 65535
int	32	–2147483648 to 2147483647	0 to 4294967295
long	32	–2147483648 to 2147483647	0 to 4294967295
float	32	+/– 1.18 x 10^{-38} to +/– 3.4 x 10^{38}	

The integral data types (**char**, **short**, **int**, and **long**) are signed by default; however, they may be used in combination with the **unsigned** keyword to yield unsigned data types (**unsigned** by itself means **unsigned int**). All integral data types use two’s complement representation.

Floating-Point Data Types

Floating-point data types are stored in the IEEE single and double precision formats. Both formats have a sign bit field, an exponent field, and a fraction field. The fields represent floating-point numbers in the following manner:

```
Floating-Point Number = <sign> 1.<fraction field> x 2(<exponent field> - bias).
```

Sign Bit Field. The sign bit field is the most significant bit of the floating-point number. The sign bit is 0 for positive numbers and 1 for negative numbers.

Fraction Field. The fraction field contains the fractional part of a "normalized" number. "Normalized" numbers are greater than or equal to 1 and less than 2. Since all normalized numbers are of the form "1.XXXXXXXX", the "1" becomes implicit and is not stored in memory. The bits in the fraction field are the bits to the right of the binary point, and they represent negative powers of 2. For example:

$0.011 \text{ (binary)} = 2^{-2} + 2^{-3} = 0.25 + 0.125 = 0.375.$

Exponent Field. The exponent field contains a biased exponent; that is, a constant bias is subtracted from the number in the exponent field to yield the actual exponent. (The bias makes negative exponents possible.)

If both the exponent field and the fraction field are zero, the floating-point number is zero.

NaN. A NaN (Not a Number) is a special value which is used when the result of an operation is undefined. For example, adding positive infinity to negative infinity results in a NaN.

Float

The **float** data type is stored in the IEEE single precision format which is 32 bits long. The most significant bit is the sign bit, the next 8 most significant bits are the exponent field, and the remaining 23 bits are the fraction field. The bias of the exponent is 127. The range of single precision format values is from 1.18 x 10^{-38} to 3.4 x 10^{38}. The floating-point number is precise to 6 decimal digits.

31	**30 23**	**22 0**
S	**Exp. + Bias**	**Fraction**

```
 0   000   0000  0 000  0000  0000  0000  0000  0000 =  0.0
 0   011   1111  1 000  0000  0000  0000  0000  0000 =  1.0
 1   011   1111  1 011  0000  0000  0000  0000  0000 =  -1.375
 1   111   1111  1 111  1111  1111  1111  1111  1111 =  NaN (Not a Number)
```

Double

The **double** data type is stored in the IEEE double precision format which is 64 bits long. The most significant bit is the sign bit, the next 11 most significant bits are the exponent field, and the remaining 52 bits are the fraction field. The bias of the exponent is 1023. The range of double precision format values is from 2.23 x 10^{-308} to 1.8 x 10^{308}. The floating-point number is precise to 15 decimal digits.

63	62 52	51 0
S	Exp. + Bias	Fraction

```
0   000   0000  0000 0000  0000  0000  ...  0000  0000  0000  0000 = 0.0
0   011   1111  1111 0000  0000  0000  ...  0000  0000  0000  0000 = 1.0
1   011   1111  1110 0110  0000  0000  ...  0000  0000  0000  0000 = -0.6875
1   111   1111  1111 1111  1111  1111  ...  1111  1111  1111  1111 = NaN
```

Precision of Real Number Operations

In the absence of the "generate code for the 68881/2" command line option, all real number operations are accomplished by calls to the real number routines (described in the "Conversion" and "Floating-Point Routines" sections of the "Run-Time Library Description" chapter) or to math library routines which eventually call run-time library routines. With the "generate code for the 68881/2" command line option, most real number operations are performed in-line with 68881/2 instructions.

All of this has a subtle effect on the precision of floating-point results.

Without the 68881/2. When routines are used to perform floating-point operations, all intermediate results are truncated to 64-bit precision immediately, and no 80-bit intermediate results are carried on into subsequent calculations. The precision of the results reflects this implementation.

With the 68881/2. When the "generate code for the 68881/2" (**-f**) command line option is used, many intermediate results are kept with 80 bits of precision and are passed on into subsequent operations without truncation. The 68881/2 itself supports a mode in which these results are automatically truncated; however, an execution speed penalty is incurred. Thus, it is important to understand, when using the "generate code for the 68881/2" command line option, that results will differ from those produced without the option.

Characters

In addition to the **char** type, the C compiler supports wide (extended) characters with the **wchar_t** type. The **wchar_t** type is implemented as **unsigned long**. Constants in the extended character set are written with a preceeding **L** modifier. Library routines which support wide characters are described under *mblen* in the "Libraries" chapter.

Multi-byte characters are not supported.

If a multi-character constant (for example, ’abc’) is encountered, the compiler multiplies the value of the first character by 256 and adds the value of the second character. If there are remaining characters, the new value is multiplied by 256 and the next character is added until no more characters are left. (Some previous versions of the compiler technology simply accepted the first character and discarded the others.)

Derived Data Types

The following objects are derived data types. The sizes of each data type (or the calculation used to determine the size) are listed.

Pointers	32-bits.
Arrays	(Number of elements)*(Size of one element).
Structures	Sum of the sizes of each member. (Members, as well as the structure itself, may be padded for alignment.)
Unions	Size of the largest member. (This member, as well as the union itself, may be padded for alignment.)
Enum types	1, 2, or 4 bytes depending on the constant values of the elements.

Pointers

Pointers are addresses which point to stored values. Pointers occupy four bytes and are aligned on four-byte boundaries (two-byte boundaries for the 68000 and 68332). The following program is a simple example of how pointers are used.

```
main()
{
        int     value;
        int     *ptr    /* "ptr" is of type pointer to "int".  */

        value = 256;
        ptr = &value;   /* "ptr" = the address of the location  */
                        /* at which "value" is stored.          */
}
```

Arrays

Arrays are made up of a fixed number of elements of the same type. Multi-dimensional arrays can be thought of as arrays of arrays (of arrays, etc.) where each array represents a single dimension. Index values for each dimension are used to access the elements of a multi-dimensional array.

The amount of storage allocated for an array is the sum of the space used by all its elements. An array is aligned on the alignment boundary of its elements. For example, a **short** array with 10 elements would use 20 bytes and be aligned on a two byte boundary.

The first element of a one-dimensional array (index equals zero) is located at the lowest address of the storage allocated for the array. Elements of multi-dimensional arrays are stored in row-major order (in other words, the rightmost index changes more rapidly with successive memory locations).

The following program shows some simple arrays.

```
float   fpns[10];       /* 10*4 = 40 Bytes of storage allocated    */
main()
{
        int     array[4][7];    /* 4*7*4 = 112 Bytes allocated     */
        int     i, j;           /* on the stack.                   */

        fpns[1] = 1.0;
        for (i = 0; i < 4; i++)
                for (j = 0; j < 7; j++)
                        array[i][j] = 0;
}
```

Strings

Strings are a sequence of characters or escape sequences enclosed in double quotes ("). Strings may be used in two distinct contexts. The first is in C program statements or as intitializers of type **char *** where they are treated as if they are of type "**const char ***". For example:

```
char    *p, *q = "abc";
p = "xyz";
```

When used in such a context, the compiler places the string, together with an additional NULL (0) termination character, in the named CONST linker section (named "const" by default).

The second context in which strings may be used is as initializers of arrays of **char**. If the initialized array is an automatic, the initialization occurs at run-time, and the compiler places the string and NULL terminator in the named CONST linker section just as above. If, however, the array being initialized is a static, the initialization occurs at load-time (or is in ROM). For example:

```
const char      string[] = "abcdefghi";
```

When a string is used to initialize an array, the compiler places the initialized array in either the named DATA linker section (if the array's type is not "**const**") or in the named CONST linker section (if the array's type is "**const**"). A terminating NULL (0) character is appended to the string only if there is room in the declared array (or if it is "open" as above).

Note Trying to change the value of a string constant may cause unwanted side effects. The reason for this is explained in the "Optimizations" chapter.

The compiler accepts hexadecimal escape sequences of unlimited length. The example below is interpreted as a single hex value:

```
*str = "\x064f";
```

In order to produce the string "df", you could modify the string in the following way:

```
*str = "\x064" "f";
```

Structures

Structures are named collections of members. Structure members may be of different types, they may be specified as bit fields, or they may even be pointers to the structure in which they are defined (self-referential structures).

Structures may be passed as parameters to and returned from functions. (See the "Stack Frame Management" section of the "Compiler Generated Assembly Code" chapter for more information on how structures are passed to and returned from functions.)

The amount of storage allocated for a structure is the sum of the space required by all its members, the alignment padding between members, and padding at the end of the structure to make its size a multiple of four (two) bytes. For example, a structure whose members are a **char**, an **int**, and a **double** would be allocated 16 bytes (three bytes following the **char** are "wasted" to align the **int**). (For the 68000 and 68332, 14 bytes are allocated—one byte following the **char** is "wasted" to align the **int**). Members are located in the allocated space in the order that they are declared.

An example of a simple structure follows.

```
struct example {       /* 16 bytes of storage allocated at 4-byte boundary.  */
        char    c;     /* First byte of structure.  */
        int     i;     /* Begins at 5th byte of structure.  */
        double  d;     /* Begins at 9th byte of structure.  */
} var;

main()
{
        var.c = ’a’;
        var.i = -1;
        var.d = 1.0;
}
```

For the 68000 and 68332 processors, **`struct example`** uses 14 bytes of storage allocated at a 2-byte boundary. **`int i`** begins at the 3rd byte of the structure and **`double  d`** begins at the 7th byte of the structure.

See the "Alignment Considerations" section for information on how the **-Q** option affects member alignment.

Bit Fields

Bit fields are structure or union members which are defined as a number of bits. A colon separates the length of a bit field from the declarator. Bit fields can be signed (declared as plain integral types) or unsigned (declared as **unsigned** integral types). All integral types are allowed in bit field declarations, but are converted to **int** or **unsigned int**. The high order bit of a signed bit field is the sign bit.

Bit fields are packed from the high-order bits to the low-order bits in the words of memory they occupy. Bit padding can be generated by omitting the name from the bit field declaration. Consecutive bit fields are packed adjacently regardless of integer boundaries. However, a bit field with a specified width of zero will cause the following bit field to start on the next **int** (double word) (**short** word for the 68000) boundary.

Examples of bit field declarations follow.

```
struct {
        int             f1:8;     /* f1 is a signed bit field,     */
                                  /* occupying bits 31-24 of the   */
                                  /* first double word.            */
                                  /*                               */
        unsigned        :12;      /* 12 bits of padding occupy     */
                                  /* bits 23-12 of the first       */
                                  /* double word.                  */
                                  /*                               */
        unsigned        f2:16;    /* f2 occupies bits 11-0 of the  */
                                  /* first double word and bits    */
                                  /* 31-28 of the second double    */
                                  /* word.                         */
                                  /*                               */
        int             :0,f3:7;  /* f3 occupies bits 31-25 of     */
                                  /* the third double word.        */
} a;                              /* The size of the structure is  */
                                  /* 12 bytes.                     */
```

Unions

Unions are like structures except that each member has a zero offset from the beginning of the union. Unions provide a way to access the same memory locations in more than one format. A simple example of a union is shown below.

```
union {
        unsigned int    sign:1;
        float           fp_rep;
} fp_num;

main()
{
        fp_num.fp_rep = 1.0;
        if (fp_num.sign == 0)
                fp_num.sign = 1;
}
```

Accessing **int** and **char** members or **int** and **short** members will yield different results due to the byte ordering of data on the 68000. See the "Byte Ordering" section which follows.

Enumeration Types

Enumeration type declarations define elements of a finite set. Each element of the enumerated type becomes a constant. The first element is equal to a constant value of 0, the second is equal to 1, and so on. You can assign a particular constant value to an element, and the values of the elements which follow will increment from that value.

An enumeration type is considered to be the smallest integral type which can represent all the values of the enumeration.

- If the constant values for all elements are between **-**128 and +127, the enumeration type is allocated the same space as **char** types.
- If the condition above is not true, but the constant values for all elements are between **-**32768 and +32767, the enumeration type is allocated the same space as **short int** types.
- If the constant value of any element is outside the range **-**32768 to +32767, the enumeration type is allocated the same space as **int** types.

An **enum** typed variable can be used in expressions wherever integral typed variables are allowed. The following program shows a simple enumerated type.

```
enum color      {yellow, red, green, blue=8, violet} paint;

/* The elements of the enumeration type "color" equal the          */
/* following constants: yellow = 0, red = 1, green = 2,            */
/* blue = 8, and violet = 9.                                       */

main()
{
        enum color      marker;

        if (marker == green)
        {
                paint = marker;
                marker++;       /* This statement is allowed, but  */
                                /* marker = 3 instead of "blue"    */
                                /* which is 8.                     */
        }
}
```

The values of an enumerated type are considered to be declared the moment they are encountered in the source file. Thus it is possible to have a declaration like the following:

```
enum {apple, orange = apple} e;
```

Alignment Considerations

Variable and constant data, as opposed to executable instructions, may be *aligned* or *padded* by the compiler. In this context, *aligned* is defined to mean that the memory allocated to the variable begins at a particular byte boundary (e.g., an alignment of four (two) bytes means that a variable’s absolute address is a multiple of four (two)); *padded* is defined to mean that the size of a type was rounded up to guarantee that the number of bytes in that type is a multiple of four (two).

Arrays are aligned according to their element type’s alignment and are not padded. Note, however, that an array’s elements may be padded (if it is an array of structures or unions).

Structure members are aligned relative to the start of the structure (and padded if they are structures or unions) in accordance with their type.

Unless function prototypes are used (see the "ANSI Extensions" section in the "C Compiler Overview" chapter), all **char** and **short** parameters are widened to **int**s when they are passed and, thus, follow **int** alignment rules when they are passed.

Note that inside a called function, **char** or **short** parameters are reduced to their normal **char** and **short** size.

Alignment can be changed by using the compiler’s "word align data" (**-Q**) option. In the presence of this option, data is aligned at word boundaries.

Note The "word align data" option is not available for the 68000 and 68332, because the compiler always word-aligns data for these processors.

The following table summarizes the default alignment and padding of the various data types when the "word align data" option is not used. The numbers in parentheses are for the 68000/10, 68332, and 68302 processors.

Table 3-2. Arithmetic Data Type Alignment

Data Type	Alignment	Padded?
char	1	N
short	2	N
int	4(2)	N
long	4(2)	N
pointer	4(2)	N
float	4(2)	N
double	4(2)	N
struct	4(2)	Y

Alignment Examples

These examples assume that the "word align data" option is not used.

Default alignment dictates that a **char** variable followed by an **int** variable "wastes" three bytes (one byte for the 68000 and 68332) of memory between the two objects. Note that there are no "wasted" bytes when a **char** variable is followed by an array of **char**, but memory is "wasted" when a **char** variable is followed by a structure.

The **sizeof** bytestruct declared with:

```
struct {char element;} bytestruct;
```

is four (two) (the minimum **sizeof** any **struct** type) and the **sizeof** biggerstruct declared with:

```
struct {char element1;
        int  element2;} biggerstruct;
```

is eight (one for element1, three "wasted" for alignment, four for element2, and none for padding as the size is a multiple of four). For the 68000 and 68332, the size is six (one for element1, one "wasted" for alignment, four for element2, and none for padding as the size is a multiple of two).

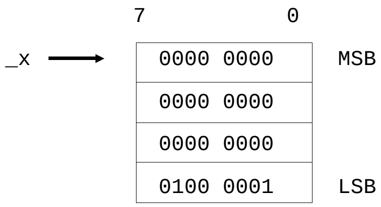
Byte Ordering

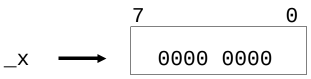
Some programs rely on byte ordering. For example, programs that declare a variable **int** in one module and **char** in another may work differently on the 68000 than on other microprocessors due to byte ordering. Consider the two files shown below:

```
File 1:                                    File 2:
-------------------------------------      --------------------------------------

extern int x;                              char x;

main()                                     funct()
{                                          {
        x = ’A’;                                   char c;
        funct();
}                                                  c = x;    /*  c = 0x00   */
                                           }                 /*  c != 0x41  */
```





Because of the problems which can be caused by relying on processor-dependent ordering, you should not write code like this. Each module or file should declare variables with identical type information.

4

Compiler Generated Assembly Code

Description of the assembly code generated by the compiler.

Chapter 4: Compiler Generated Assembly Code

The compiler generates assembly code for the HP B3641 assembler (as68k). Knowing how the compiler generates this code will help you to write assembly language routines that interface with C functions.

In this chapter you will find information about the following subjects:

- Assembly language symbol names
- Debug directives
- Stack frames (how parameters are passed to and from C functions)
- Register usage
- Run-time error checking
- Ways to include assembly language in a C source file

Assembly Language Symbol Names

The compiler prefixes characters to the names given in the C source (to prevent potential conflicts with assembler reserved words) when generating assembly language symbols to represent addresses and stack offsets of C variables.

Symbol Prefixes

The _ Prefix

Externs, globals, statics, and functions have an underscore (_) prefix. You can change the prefix for external variables (externs, globals, and functions) to a different string by using a cc68k option (**-Wc,-l**). Refer to the on-line man page for more information on changing this prefix character.

The S_ Prefix

Parameters and automatics have "S_" prefixed. The "S" indicates symbols that are SET equal to stack offsets.

The L_ Prefix

The only other symbol names from the C source which are passed on to the assembly code are C label names. These labels have "L_" and a unique ASCII number prefixed to them in the generated assembly code.

See figure 4-1 for an example of how the compiler creates symbol names.

These symbol names are *not* used by debuggers and emulators unless the debuggers and emulators consume HP format absolute files. The C source symbol names are defined using debug directives (see the following "Debug Directives" section).

```
                                /*   Assembly Symbol Name:     */
                                /*   ---------------------     */
float   ext_var;                /*      _ext_var               */
                                /*                             */
main()                          /*      _main                  */
{                               /*                             */
        char        auto_var;   /*      S_auto_var             */
        static int  number;     /*      _1_number              */
                                /*                             */
        auto_var = ’a’;         /*                             */
        goto label;             /*                             */
label:                          /*      L_2_label              */
        function(number);       /*                             */
}                               /*                             */
                                /*                             */
int     number;                 /*      _number                */
                                /*                             */
function(i)                     /*      _function              */
int i;                          /*      S_i                    */
{                               /*                             */
        i = 1;                  /*                             */
}                               /*                             */
```

Figure 4-1. Examples of Generated Symbol Names

Situations Where C Symbols are Modified

There are four cases where the compiler modifies the names of C variables to guarantee that they are unique in the assembly code:

1 If a parameter or automatic name exceeds 29 characters in length, then it must be made unique since the assembler only recognizes 31 (29 + 2 for "S_") significant characters in a symbol.

2 If there is a variable with the same name in a containing scope in the C source, then a parameter or automatic name must be made unique since both symbols must exist at the same time in the assembler (which doesn’t understand scoping).

3 All local statics (those declared inside a function) are made unique, since a global static of the same name may be declared later.

4 External statics (those declared outside a function) are made unique if their name exceeds 30 characters in length since the assembler only recognizes 31 (30 + 1 for "_") significant characters in a symbol.

In all four cases, symbol names are made unique by inserting a unique ASCII number and an underscore between the initial underscore (or "S_") and the C name. For example:

```
_123_name
S_123_name
```

#pragma ALIAS

Syntax:

#pragma ALIAS *Csymbolname Assemsymbolname*
#pragma ALIAS *Csymbolname "Assemsymbolname"*

This pragma allows overriding of the C compiler algorithm for converting C source file symbol names into an unique assembler symbol names (the algorithm generally prefixes an "_" or "S_"). This pragma should be **used with great care** as it may generate assembly-time errors due to conflicts between *Assemsymbolname* and other assembly language symbols. Use the quotation marks if the *Assemsymbolname* would not be a valid C identifier. This pragma should be placed before any references to the symbol.

Compiler Generated Symbols

The compiler generates assembly language labels for C loops, switch statements, and other constructs which require labels. The name of the label is related to the use of the label; for example, the label "forLoop3" might be used to implement a **`for`** loop.

Debug Directives

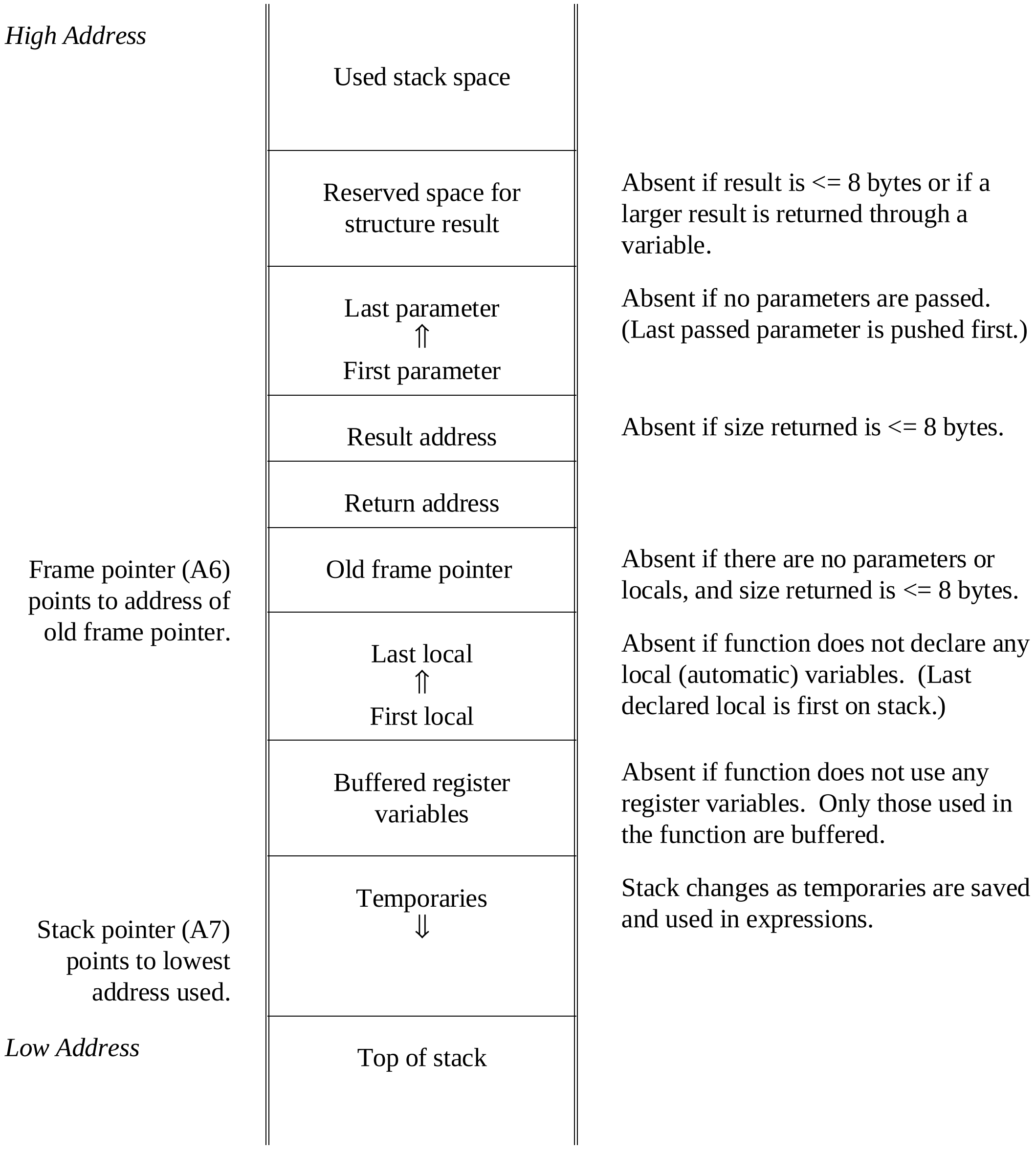
If the "strip symbol table information" compiler command line option is not used, the compiler generates all the HP B3641 debug directives necessary to use debugger, emulation, and analysis tools. This debug information consists of source file and line references, type names and structure, symbol type and access information, and function call information. One LINE directive is output for each C source statement to associate the generated assembly code with the C source file line number.

Stack Frame Management

In block-structured languages (C, Pascal, etc.), the stack is used to pass parameters into and receive results from each of the blocks which make up the program. In C, these blocks are called functions. In addition to passing values and returning results, the stack is used for a function’s local variables and to buffer register variables. The area of the stack used by a function is called a "stack frame". To illustrate what makes up stack frames and how they are managed, one must observe what happens to the stack when a function is called; these events are listed below and described in this section.

Note

This section applies only to C function calls. Run-time libraries invoked in compiler-generated code may use different (and more efficient) stack frame management because these calls are not constrained by C language calling conventions.



High Address	Used stack space	
	Reserved space for structure result	Absent if result is <= 8 bytes or if a larger result is returned through a variable.
	Last parameter ⇑ First parameter	Absent if no parameters are passed. (Last passed parameter is pushed first.)
	Result address	Absent if size returned is <= 8 bytes.
	Return address	
Frame pointer (A6) points to address of old frame pointer.	Old frame pointer	Absent if there are no parameters or locals, and size returned is <= 8 bytes.
	Last local ⇑ First local	Absent if function does not declare any local (automatic) variables. (Last declared local is first on stack.)
	Buffered register variables	Absent if function does not use any register variables. Only those used in the function are buffered.
Stack pointer (A7) points to lowest address used.	Temporaries ⇓	Stack changes as temporaries are saved and used in expressions.
Low Address	Top of stack	

Figure 4-2. Stack Frame Format

- Space is reserved for a structure result (if the size returned is greater than 8 bytes).
- Parameters are pushed (last is pushed first).
- A pointer to the result address is pushed (if size returned is greater than 8 bytes).
- The subroutine call is made and the return address is pushed (with the JSR or BSR instructions).
- The old frame pointer is pushed (with the LINK instruction).
- Space for automatics (locals) is allocated (also by LINK).
- Registers used in the called function for register variables are pushed (to buffer their values).
- During function execution, intermediate values may be stored on the stack temporarily.
- Function return values are stored in working registers or returned indirectly through a pointer on the stack (possibly into space reserved on the stack).
- At function exit, register variables are restored and locals are deallocated; and the calling routine deallocates parameters and uses the structure result.

The general format of a stack frame is shown in figure 4-2. An example of the code generated for stack frame management is shown in figure 4-3.

Structure Results

C allows functions to return results of type **struct**. Although most function results are returned in register D0, D1, or FP0, structures greater in size than 8 bytes are returned to a location specified by the result location pointer. The result location pointer is pushed onto the stack after the parameters and before the return address.

In a C statement such as "**structure = f(x)**", the address of the variable "structure" may be pushed as the result location pointer, and the called function will return its resultant structure directly into memory reserved for the "structure" variable.

In other statements, such as "**i = f(x).field**", space must be reserved on the stack (prior to pushing parameters) to hold the function structure result. The address of this reserved stack space will be pushed as the result location pointer (after the parameters and before the return address), and the function will return its resultant structure into the reserved stack space. **This approach maintains reentrancy for functions returning structures**.

Parameter Passing

Parameters are pushed on the stack in right to left order as they appear in the function call (in other words, the last passed parameter is pushed first). Unless function prototypes are used (see the "ANSI Extensions" section in the "C Compiler Overview" chapter), parameters of type **char** and **short** are rounded up to **int** when passed, and parameters of type **float** are rounded up to **double** when passed.

After the parameters (and, possibly, a result address) are pushed, the function is called. The subroutine call pushes the return address on the stack following the parameters.

Pushing the Old Frame Pointer and Allocating Space

A LINK is the first instruction executed inside a called function. The LINK instruction will:

1. Push the old frame pointer (register A6).
2. Load A6 with the current value of the stack pointer. (A6 becomes the new frame pointer.)
3. Decrement the stack pointer (register A7) by the amount of space required for all local (automatic) variables used by the function. (Total local space is padded to a multiple of two bytes.)

The LINK (and UNLINK at function exit) are optimized out whenever a function has no parameters, no automatics, and returns a result of size eight bytes or less.

Buffering Registers Used for Register Variables

Following the allocation of automatics, any registers which have been allocated for use by the function as register variables are pushed on the stack to buffer their values.

Also, the compiler may use these registers for automatics regardless of whether or not they have been declared with the **register** storage class specifier (see the "Register Usage" section which follows). Any registers used by the function for automatics are also pushed on the stack.

Accessing Parameters

Each parameter's assembly language symbol name is SET to that parameter's offset from the frame pointer. The offset of the first parameter will be 8 if the result size is less than or equal to 8 bytes; the offset of the first parameter will be 12 if the result size is greater than 8 bytes. For example, if "p" is the first parameter passed, the compiler generates the following line in the assembly:

```
S_p      SET     8
```

Parameters are accessed by using the symbol names relative to A6. For example:

```
         MOVE.L  (S_p,A6),D0
```

Shortening Parameters

Unless function prototypes are used (see the "ANSI Extensions" section in the "C Compiler Overview" chapter), parameters of type **char** and **short** are widened to **int** when passed. Thus, any parameters formally declared to be of type **char** or **short** must be shortened from **int**. Since this shortening is defined to be by truncation, it is accomplished by simply adjusting the **int** parameter's offset from the frame pointer to point to the least significant word (**short**) or byte (**char**).

Similarly, **float** parameters are widened to double when passed. Thus, any formal **float** parameters must be shortened from their passed **double** form. To avoid problems when such parameters are optional, a **float** local variable is allocated, and the **double** value is converted to **float** and stored in the local variable. The formal parameter's offset from the frame pointer is then set to be that of the new local variable.

An example of the widening and shortening of parameters is shown in figure 4-4. The same example using function prototypes is shown in figure 4-5.

Accessing Locals

The last local (automatic) variable declared appears first on the stack. Each local variable's assembly language symbol name is SET to that variable's offset from the frame pointer. For example, if "l" is the first local declared, and there are 20 bytes of local variables, then the compiler generates the following line in the assembly:

```
S_l     SET     -20
```

Local variables are accessed using the symbol name relative to A6. For example:

```
        MOVE.L  (S_l,A6),D0
```

Using the Stack for Temporary Storage

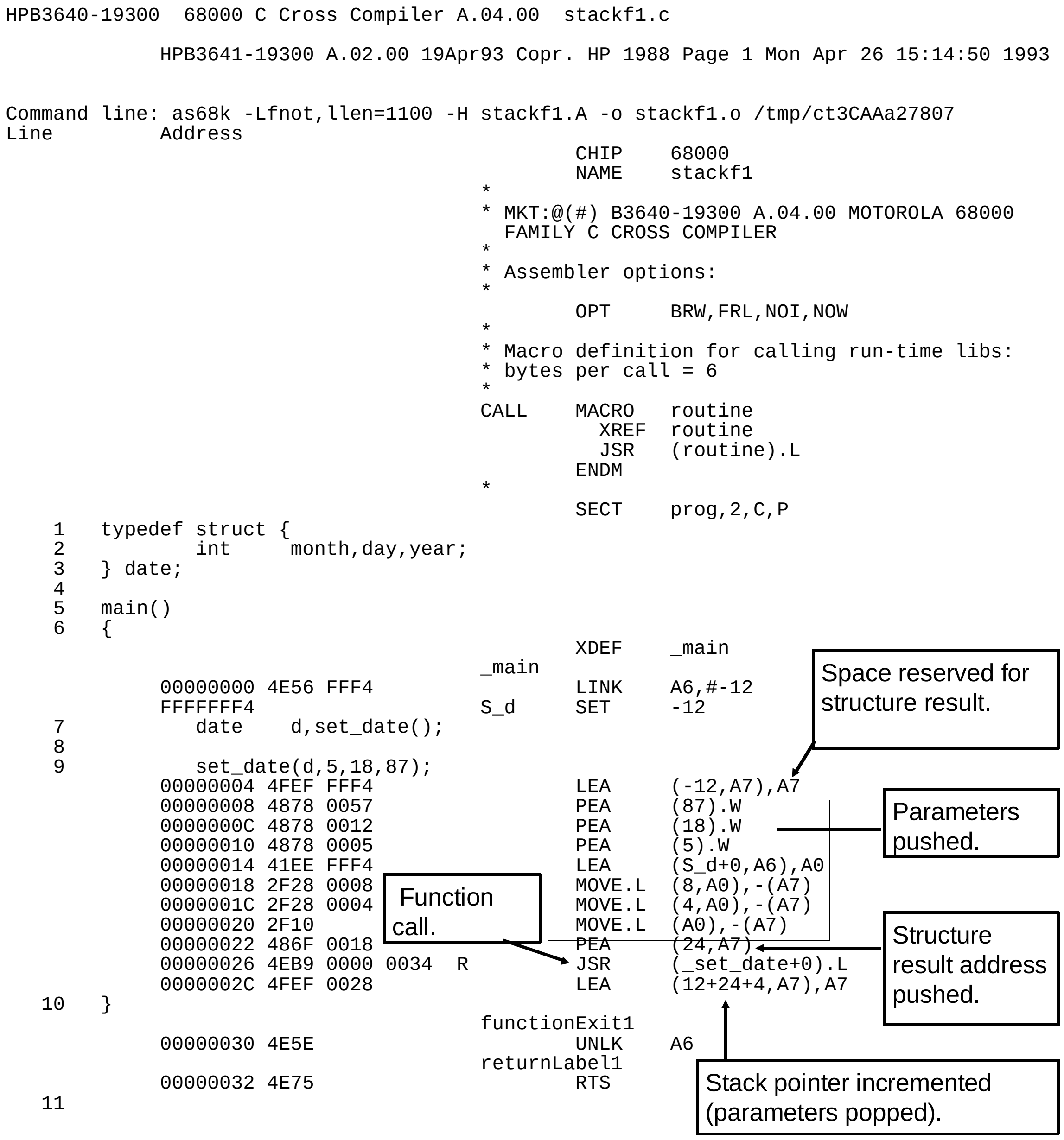
Code generated by the function body may or may not use the stack for temporary storage of intermediate results. This temporary storage size is dynamic through the function, but has all been removed by the time the function exit code is executed.

Function Results

Results which fit in four bytes are returned in register D0. Results of four to eight bytes are returned in registers D0 and D1. Results larger than eight bytes are returned indirectly through a "result address" pointer pushed by the calling routine. This pointer may point to a static memory location, an automatic variable, or temporary space on the stack. For the 68040, functions return float or double values in the FP0 register.

Function Exit

At function exit, any buffered register variables are popped, an UNLINK instruction is used to restore the buffered frame pointer and increment the stack pointer to its position at function entry, and the function return itself pops the return address. The calling routine is responsible for incrementing the stack pointer, popping the passed parameters, and, if necessary, removing the space reserved for structure function results.



```
HPB3640-19300  68000 C Cross Compiler A.04.00  stackf1.c

             HPB3641-19300 A.02.00 19Apr93 Copr. HP 1988 Page 1 Mon Apr 26 15:14:50 1993


Command line: as68k -Lfnot,llen=1100 -H stackf1.A -o stackf1.o /tmp/ct3CAAa27807
Line         Address
                                                CHIP    68000
                                                NAME    stackf1
                                        *
                                        * MKT:@(#) B3640-19300 A.04.00 MOTOROLA 68000
                                          FAMILY C CROSS COMPILER
                                        *
                                        * Assembler options:
                                        *
                                                OPT     BRW,FRL,NOI,NOW
                                        *
                                        * Macro definition for calling run-time libs:
                                        * bytes per call = 6
                                        *
                                        CALL    MACRO   routine
                                                  XREF  routine
                                                  JSR   (routine).L
                                                ENDM
                                        *
                                                SECT    prog,2,C,P
    1   typedef struct {
    2           int     month,day,year;
    3   } date;
    4
    5   main()
    6   {
                                                XDEF    _main
                                        _main
             00000000 4E56 FFF4                 LINK    A6,#-12
             FFFFFFF4                   S_d     SET     -12
    7           date    d,set_date();
    8
    9           set_date(d,5,18,87);
             00000004 4FEF FFF4                 LEA     (-12,A7),A7
             00000008 4878 0057                 PEA     (87).W
             0000000C 4878 0012                 PEA     (18).W
             00000010 4878 0005                 PEA     (5).W
             00000014 41EE FFF4                 LEA     (S_d+0,A6),A0
             00000018 2F28 0008                 MOVE.L  (8,A0),-(A7)
             0000001C 2F28 0004                 MOVE.L  (4,A0),-(A7)
             00000020 2F10                      MOVE.L  (A0),-(A7)
             00000022 486F 0018                 PEA     (24,A7)
             00000026 4EB9 0000 0034  R         JSR     (_set_date+0).L
             0000002C 4FEF 0028                 LEA     (12+24+4,A7),A7
   10   }
                                        functionExit1
             00000030 4E5E                      UNLK    A6
                                        returnLabel1
             00000032 4E75                      RTS
   11
```

Space reserved for structure result.

Parameters pushed.

Function call.

Structure result address pushed.

Stack pointer incremented (parameters popped).

Figure 4-3. Example Stack Frame Management Code

```
12   date set_date(x,mo,da,yr)
13   date    x;
14   int     mo,da,yr;
15   {
                                             XDEF    _set_date
                                     _set_date
          00000034 4E56 FFF0                 LINK    A6,#-16
          0000000C                   S_x     SET     12
          00000018                   S_mo    SET     24
          0000001C                   S_da    SET     28
          00000020                   S_yr    SET     32
          FFFFFFF0                   S_i1    SET     -16
          FFFFFFF8                   S_i2    SET     -8
                                     *
                                     *
16         double          i1,i2;
17         register int    i3;
18
19         x.month = mo;
          00000038 2D6E 0018 000C            MOVE.L  (S_mo+0,A6),(S_x+0,A6)
20         x.day = da;
          0000003E 2D6E 001C 0010            MOVE.L  (S_da+0,A6),(S_x+4,A6)
21         x.year = yr;
          00000044 2D6E 0020 0014            MOVE.L  (S_yr+0,A6),(S_x+8,A6)
22         return(x);
          0000004A 41EE 000C                 LEA     (S_x+0,A6),A0
          0000004E 226E 0008                 MOVEA.L (8,A6),A1
                                             REPT    3
                                             MOVE.L  (A0)+,(A1)+
                                             ENDR
          00000052 22D8                      MOVE.L  (A0)+,(A1)+
          00000054 22D8                      MOVE.L  (A0)+,(A1)+
          00000056 22D8                      MOVE.L  (A0)+,(A1)+
23   }
                                     functionExit2
          00000058 4E5E                      UNLK    A6
                                     returnLabel2
          0000005A 4E75                      RTS
                                             END
```

Old frame pointer pushed and space for locals allocated.

Structure result returned.

Function exit.

Figure 4-3. Example Stack Frame Mgmt. Code (Cont'd)

```
HPB3640-19300  68000 C Cross Compiler A.04.00  shrtwid1.c

             HPB3641-19300 A.02.00 19Apr93 Copr. HP 1988 Page 1 Mon Apr 26 15:14:59 1993


Command line: as68k -Lfnot,llen=1100 -H shrtwid1.A -o shrtwid1.o /tmp/ct3CAAa27822
Line         Address
                                                CHIP    68000
                                                NAME    shrtwid1
                                        *
                                        * MKT:@(#) B3640-19300 A.04.00 MOTOROLA 68000
FAMILY C CROSS COMPILER
                                        *
                                        * Assembler options:
                                        *
                                                OPT     BRW,FRL,NOI,NOW
                                        *
                                        * Macro definition for calling run-time
libraries:
                                        * bytes per call = 6
                                        *
                                        CALL    MACRO   routine
                                                  XREF  routine
                                                  JSR   (routine).L
                                                ENDM
                                        *
                                                SECT    prog,2,C,P
    1   main()
    2   {
                                                XDEF    _main
                                        _main
             00000000 2F03                      MOVE.L  D3,-(A7)
             00000002 2F02                      MOVE.L  D2,-(A7)
                                        *
                                        * Register 'D3' is register variable 'S_c'.
                                        *
                                        *
                                        * Register 'D2' is register variable 'S_f'.
                                        *
    3           char    c, char_funct();
    4           float   f, float_funct();
    5
    6           char_funct(c);
             00000004 1003                      MOVE.B  D3,D0
             00000006 4880                      EXT.W   D0
             00000008 48C0                      EXT.L   D0
             0000000A 2F00                      MOVE.L  D0,-(A7)
             0000000C 4EB9 0000 002E  R         JSR     (_char_funct+0).L
             00000012 588F                      ADDQ.L  #4,A7
    7           float_funct(f);
             00000014 2002                      MOVE.L  D2,D0
                                                CALL    ftod
                                                  XREF  ftod
             00000016 4EB9 0000 0000  E           JSR   (ftod).L
             0000001C 2F01                      MOVE.L  D1,-(A7)
             0000001E 2F00                      MOVE.L  D0,-(A7)
```

char widened to **int**.

Figure 4-4. Widening and Shortening of Parameters

```
          00000020 4EB9 0000 003A  R         JSR     (_float_funct+0).L
          00000026 508F                      ADDQ.L  #8,A7
 8   }
                                     functionExit1
          00000028 241F                      MOVE.L  (A7)+,D2
          0000002A 261F                      MOVE.L  (A7)+,D3
                                     returnLabel1
          0000002C 4E75                      RTS
 9
10   char char_funct(chr)
11   char    chr;
12   {
                                             XDEF    _char_funct
                                     _char_funct
          0000000B                   S_chr   SET     11
13          chr = ’A’;
          0000002E 1F7C 0041 0007            MOVE.B  #65,(S_chr-4,A7)
14          return(chr);
          00000034 102F 0007                 MOVE.B  (S_chr-4,A7),D0
15   }
                                     functionExit2
                                     returnLabel2
          00000038 4E75                      RTS
16
17   float float_funct(flt)
18   float   flt;
19   {
                                             XDEF    _float_funct
                                     _float_funct
          0000003A 4E56 FFFC                 LINK    A6,#-4
          FFFFFFFC                   S_flt   SET     -4
          00000008                   S_wide_param1   SET     8
          0000003E 43EE 0008                 LEA     (S_wide_param1+0,A6),A1
          00000042 2019                      MOVE.L  (A1)+,D0
          00000044 2219                      MOVE.L  (A1)+,D1
                                             CALL    dtof
                                               XREF  dtof
```

int shortened to **char** (offset points to least significant byte of parameter.)

Figure 4-4. Widening/Shortening Parameters (Cont’d)

```
HPB3640-19300  68000 C Cross Compiler A.04.00  funcprto.c

             HPB3641-19300 A.02.00 19Apr93 Copr. HP 1988 Page 1 Mon Apr 26 15:15:10 1993


Command line: as68k -Lfnot,llen=1100 -H funcprto.A -o funcprto.o /tmp/ct3CAAa27837
Line Address
                                        CHIP    68000
                                        NAME    funcprto
                                *
                                * MKT:@(#) B3640-19300 A.04.00 MOTOROLA 68000 FAMILY C
CROSS COMPILER
                                *
                                * Assembler options:
                                *
                                        OPT     BRW,FRL,NOI,NOW
                                *
                                * Macro definition for calling run-time libraries:
                                * bytes per call = 6
                                *
                                CALL    MACRO   routine
                                          XREF  routine
                                          JSR   (routine).L
                                        ENDM
                                *
                                        SECT    prog,2,C,P
    1   main()
    2   {
                                        XDEF    _main
                                _main
     00000000 2F02                      MOVE.L  D2,-(A7)
                                *
                                * Register 'D0' is register variable 'S_c'.
                                *
                                *
                                * Register 'D2' is register variable 'S_f'.
                                *
    3           char    c, char_funct(char);
    4           float   f, float_funct(float);
    5
    6           char_funct(c);
     00000002 1F00                      MOVE.B  D0,-(A7)
     00000004 4EB9 0000 0018  R         JSR     (_char_funct+0).L
    7           float_funct(f);
     0000000A 2F02                      MOVE.L  D2,-(A7)
     0000000C 4EB9 0000 0024  R         JSR     (_float_funct+0).L
     00000012 5C8F                      ADDQ.L  #6,A7
    8   }
                                functionExit1
     00000014 241F                      MOVE.L  (A7)+,D2
                                returnLabel1
     00000016 4E75                      RTS
    9
   10   char char_funct(
```

char no longer widened to **int**.

Figure 4-5. Function Prototype Parameter Passing

```
11   char    chr)
12   {
                                     XDEF    _char_funct
                             _char_funct
  00000008                   S_chr   SET     8
13          chr = ’A’;
  00000018 1F7C 0041 0004            MOVE.B  #65,(S_chr-4,A7)
14          return(chr);
  0000001E 102F 0004                 MOVE.B  (S_chr-4,A7),D0
15   }
                             functionExit2
                             returnLabel2
  00000022 4E75                      RTS
16
17   float float_funct(float flt)
18   {
                                     XDEF    _float_funct
                             _float_funct
  00000008                   S_flt   SET     8
19          flt = 1.0;
  00000024 2F7C 3F80 0000            MOVE.L  #$3F800000,(S_flt-4,A7)
           0004
20          return(flt);
  0000002C 202F 0004                 MOVE.L  (S_flt-4,A7),D0
21   }
                             functionExit3
                             returnLabel3
  00000030 4E75                      RTS
                                     END
```

Figure 4-5. Function Prototype Parameters (Cont’d)

Register Usage

The following table shows how registers are used for C function calls.

Note

This section applies only to C function calls. Run-time libraries invoked in compiler-generated code may use other conventions understood by the calling code. (See the "Run-Time Library Description" chapter.)

Table 4-1. Register Usage

<table>
<tr><th>Register</th><th colspan="2">Register contents</th></tr>
<tr><td><b>D0, D1</b></td><td>Return values</td><td rowspan="3">Working registers</td></tr>
<tr><td><b>A0, A1</b></td><td></td></tr>
<tr><td><b>FP0, FP1</b></td><td></td></tr>
<tr><td><b>A5</b></td><td colspan="2">Reserved for A5 relative addressing</td></tr>
<tr><td><b>A6</b></td><td colspan="2">Frame pointer</td></tr>
<tr><td><b>A7</b></td><td colspan="2">Stack pointer</td></tr>
</table>

Registers D0, D1, A0, and A1 are reserved as working registers to hold intermediate values of calculations. Registers D0 and D1 are also used to hold function return values that fit in eight bytes. Return values larger than eight bytes are returned indirectly via a pointer.

For more information on how register A5 is used for A5 relative addressing, see the "Addressing Modes" section in the "Embedded Systems Considerations" chapter.

When the "generate 68881/2 code" (**-f**) option is used, registers FP2-FP7 are reserved for floating-point register variables. The compiler may assign **float** or **double** objects to these registers. Also, when the "generate 68881/2 code" option is used, registers FP0 and FP1 are reserved as working registers.

For the 68040, registers FP2-FP7 are reserved for floating-point register variables. The compiler may assign **float** or **double** objects to these registers. Registers FP0 and FP1 are reserved as working registers. FP0 is also used for function return values of type **float** or **double**.

Register variables

Using the priority listed below, the compiler allocates the following types of objects to registers D2-D7, A2-A4, and FP2-FP7:

1 Variables declared with **register** storage class in the order declarations are encountered.

2 Local non-static and function parameter variables, and addresses of static variables, according to frequency of occurrence of the variable’s name in the function.

If the type of the object being declared is a pointer, the compiler prefers to assign the object to an address register; however, the compiler may assign the object to a data register if no address registers remain. The compiler only assigns non-pointers (small enough to fit in a register) to data registers.

When the "optimize" option is specified, the peephole optimizer will reallocate register variables to unused working registers. This optimization saves the "push" and "pop" instructions used to buffer register variable registers.

Function parameter names and static variable names must be used at least three times in the function before they will be considered for register allocation. The rationale for this restriction has to do with the added generated assembly instruction required to move a static or a function parameter into a register. The space cost of the added instruction is considered to be offset when three or more references are made to the parameter because now the references are to a register instead of the stack. However, it is difficult to know if the register-for-a-parameter optimization will improve execution speed because it is impossible to know how the parameter is actually used in the function body. There could be instances where this optimization could result in slower code due to the extra assignment.

Example

To better understand the allocation scheme, consider the following example. Suppose there was a single register left to allocate. A local non-static variable appears just once in the function body. A parameter appears twice in the function body. Which gets the register? The local variable does because the parameter, which appears *less than three times*, has not "qualified" for consideration for frequency of occurrence.

Now let us suppose that the parameter appears *n* times where *n* is three or greater. Suppose the local non-static variable appears *n–1* times. Which gets the register? The parameter gets the register because it has "qualified" for consideration and has a greater number of occurrences.

Run-Time Error Checking

Specifying the "generate run-time error checking" (**-g**) option causes the compiler to generate code for the following types of additional run-time error checking:

- Dereferences of all NULL pointers and uninitialized automatic pointers are detected and reported. (Dereferencing is also called *indirection*; in other words, it is access to the object to which a pointer points.) This requires the initialization of automatic pointers at run-time with a value (–1) indicating they are uninitialized. Note that static variables are not initialized to the uninitialized pointer value, because the default value for static variables is zero.

- Array references outside declaration index bounds are detected and reported.

The "generate run-time error checking" option will override the "optimize" and "strip symbol table information" options. See the on-line man pages for more information on the compiler command line options.

Using Assembly Language in the C Source File

The C compiler provides three mechanisms to embed assembly language instructions. Which one you choose depends on where you want the assembly language to appear and your purpose for including the assembly language instructions. The mechanisms are:

- **#pragma ASM** and **#pragma END_ASM**
- **__asm ("**C_string**")**
- **#pragma FUNCTION_ENTRY** "C_string",
  #pragma FUNCTION_EXIT "C_string", and
  #pragma FUNCTION_RETURN "C_string"

The compiler changes the names of C variables and functions into assembly language symbols. If you know how the changed symbol names will appear in the generated assembly code, you may easily use C variables and functions in your embedded assembly code. (For more information on symbol names, see the "Symbol Names" section in this chapter.)

When you embed assembly language, all assumptions about working registers (D0, D1, A0, A1, and FP0-FP1) for optimization purposes are forgotten.

Register variables (D2-D7, A2-A4, and FP2-FP7), A5, the frame pointer (A6), and the stack pointer (A7) are not buffered prior to embedded assembly language instructions. You should buffer these registers if they will be used in your assembly code.

Optimizations do not affect your embedded assembly code.

None of these mechanisms are part of the ANSI standard, so programs which use embedded assembly language may not be portable to other compilers.

#pragma ASM
#pragma END_ASM

Syntax:

```
#pragma ASM
       .
     (assembly language statement(s))
       .
#pragma END_ASM
```

These two pragmas bracket a portion of inline assembly code. You may use these pragmas anywhere a C statement or external declaration can occur. Place the **#pragma ASM** before the beginning of your embedded assembly code and place the **#pragma END_ASM** after the code.

The assembly instructions must conform to the format and syntax required by the HP B3641 assembler. The C compiler does not check the embedded assembly instructions for correctness. The compiler simply passes the assembly language statements, unchanged, to the assembler. You may, however, use the C preprocessor to alter embedded assembly language instructions.

Example Figure 4-6 gives an example of using the **#pragma ASM/END_ASM** to embed assembly code in a C source file.

```
main ()
{
        .
        .
        .
/* Invoke supervisor mode routine.  */

#pragma ASM
        TRAP    #17
#pragma END_ASM
        .
        .
        .
}

swap (int i1, int i2) {
#pragma ASM
        move.l  (S_i1,a6),d0
        move.l  (S_i2,a6),(S_i1,a6)
        move.l  d0,(S_i2,a6)
#pragma END_ASM
}
```

Figure 4-6. #pragma ASM/END_ASM Embedded Assembly

__asm ("C_string")

Syntax:

```
__asm ("C_string")
```

The quotes are part of the *C_string* argument and the two preceding underscores are required to meet ANSI name space requirements.

The **__asm** function is another way to embed assembly code. It differs from the **#pragma ASM/END_ASM** pair in two ways:

- **#pragma ASM/END_ASM** brackets a section of inline assembly code. In contrast, the assembly language instructions are contained in a "C_string" argument to the **__asm** function.
- **#pragma ASM/END_ASM** may appear either inside or outside of a function body. Because **__asm** is syntactically a function call, it may only appear inside a function body just as any other function call must.

The **__asm** function has some advantages over the **#pragma ASM/END_ASM** mechanism. First, this function can be part of a macro definition which means you may define a macro that contains embedded assembly language. The **#pragma ASM/END_ASM** pair cannot be used to do this. Second, for single assembly instructions, the **__asm** function is more expedient because it requires just the function call on a single line.

The "C_string" argument is a character string containing one or more lines of assembly code. (The quotes are part of the argument.) It must contain white space so that when the string is output to the generated assembly code, it will conform to the format and syntax required by the HP B3641 Assembler. The C compiler does not check the C_string for correctness. The compiler simply outputs the string to the assembly code.

Example

Figure 4-7 gives an example of using the **__asm** function.

```
main ()
{
        .
        .
        .
/* Invoke supervisor mode routine. */

__asm("\tTRAP\t#17")        .
        .
        .
}

swap (int i1, int i2) {
/* Notice the "\t" whitespace that must appear
   in order to conform to the Assembler requirement
   that instructions cannot begin in column 1. Spaces
   or a tab character would also have worked. Notice also
   that there is no need to terminate the string with a newline. */

        __asm ("\tmove.l  (S_i1,a6),d0");
        __asm ("\tmove.l  (S_i2,a6),(S_i1,a6)");
        __asm ("\tmove.l  d0,(S_i2,a6)");


/* Another, less readable way of doing the above
   involves using newlines to achieve line breaks
   in the output assembly when the C_string contains
   more than one assembly instruction.
        __asm ("\tmove.l  (S_i1,a6),d0\n\tmove.l (S_i2,a6),(S_i1,a6)");
        __asm ("\tmove.l  d0,(S_i2,a6)");

   This form would appear the same as the first
   in the output assembly code.                   */
}
```

Figure 4-7. __asm Function Embedded Assembly

#pragma FUNCTION_ENTRY, #pragma FUNCTION_EXIT, #pragma FUNCTION_RETURN

Syntax:

```
#pragma FUNCTION_ENTRY "C_string"
#pragma FUNCTION_EXIT "C_string"
#pragma FUNCTION_RETURN "C_string"
```

The third mechanism is **#pragma FUNCTION_ENTRY /EXIT /RETURN**. These pragmas are not a pair like **#pragma ASM/END_ASM**. They may be used independently of each other or they may be used together.

#pragma FUNCTION_ENTRY may be used to insert assembly language instructions into function entry code. Similarly, **#pragma FUNCTION_EXIT** and **#pragma FUNCTION_RETURN** may be used to insert assembly language instructions into function exit code. Neither **#pragma ASM/END_ASM** nor the **__asm** function is able to place embedded assembly in the function entry or exit code. The embedded code is placed is as follows:

- **#pragma FUNCTION_ENTRY** places the embedded assembly code immediately after the label generated from the function name. Because the embedded assembly occurs before any function entry code, you can modify the way a function is entered.
- **#pragma FUNCTION_EXIT** places the embedded assembly immediately *before* the function return label. That is, it follows the function exit code, but precedes the function return. (Some NOPs may appear between the embedded assembly code and the return label.) This pragma gives you the flexibility to control function return and also allows you to perform extra instructions before function return.
- **#pragma FUNCTION_RETURN** places the embedded assembly immediately *after* the function return label. Use this pragma if you want to use your own function return code. For example, you might want to trap to a debugging routine.

Remember, you may use **#pragma FUNCTION_ENTRY, FUNCTION_EXIT,** and **FUNCTION_RETURN** by themselves, or you may use all of them together.

Two limitations apply to these pragmas:

- **#pragma FUNCTION_ENTRY**, **#pragma FUNCTION_EXIT**, and **#pragma FUNCTION_RETURN** may only appear outside of a function body.
- **#pragma FUNCTION_ENTRY**, **#pragma FUNCTION_EXIT**, and **#pragma FUNCTION_RETURN** must precede the function they are to affect. They are in effect only for the immediately following function and no other.

These pragmas take a "C_string" argument. (The quotes are part of the argument and no parentheses surround the argument.) As with the **__asm** function, the "C_string" argument is a character string containing assembly language instructions. It must contain white space and newlines ("\n") so that when the string is output to the generated assembly code, it will conform to the format and syntax required by the HP B3641 assembler. The C compiler does not check the C_string for correctness. The compiler simply outputs the string to the assembly code.

Example

Figure 4-8 gives an example of using **#pragma FUNCTION_EXIT** along with **#pragma INTERRUPT** (discussed in the "Embedded Systems Considerations" chapter) to cause an interrupt service routine to trap back to the operating system instead of allowing it to terminate with an RTE instruction as it would if **#pragma INTERRUPT** were used alone. When this routine enters its function exit code, it will do the cleanup of the stack and other chores in preparation of the RTE. But because the **#pragma FUNCTION_EXIT** code causes the routine to trap back to the operating system, it will never execute the RTE.

```
#pragma INTERRUPT
#pragma FUNCTION_EXIT "\tMOVE #10,D0\n\tTRAP #12"
void intr_funct (void)
{
        .
        .
        .
/* Function body for interrupt routine. */
        .
        .
        .
}

/* The following exit code results from these
   pragmas. */

                                functionExit1
     00000004 4CDF 6303            MOVEM.L   (A7)+,D0/D1/A0/A1/A5/A6
     00000008 31C0 000A            MOVE   #10,D0
     0000000C 4E4C                 TRAP   #12
                                returnLabel1
     0000000E 4E73                 RTE
                                   END
```

Figure 4-8. #pragma FUNCTION_EXIT

Assembly Language in Macros

To use assembly language in a macro, use the **__asm** function. The **#pragma** mechanism does not work in a macro.

When you write the macro, remember the following suggestions:

- Use **__asm**, not one of the pragmas.
- Do not use macro parameters in the assemly code. The C preprocessor does not expand names inside the quotation marks.
- Use spaces and tabs (entered as "\t") to place "white space" in the assembly code.
- If you need to place more than one line of assembly language in the macro, either use an **__asm** statement for each line or place a "\n" between lines. The C preprocessor will place the entire macro on one line, then the compiler will change the "\n" to a newline when generating the assembly code.
- Be careful about changing the values of C variables (side effects) in the macro. You may wish to include the names of such variables in the name of the macro.
- You can examine the generated assembly code by compiling with **`cc68k -SL`** and looking at the **.O** file. If you need to understand how the C preprocessor affected the code, use **`cc68k -E`**.

5

Optimizations

Description of optimizations performed by the compiler.

The C compiler performs many optimizations automatically; there is also an "optimize" command line option (**-O**) to cause peephole optimization, time or space optimization, and other compile-time costly optimizations. This chapter first describes the optimizations which are always performed; next, it describes the optimizations which occur as a result of the "optimize" command line option.

Universal Optimizations

The C compiler automatically performs many optimizations on C programs. Several of the most notable types of optimizations are listed below and described in this section.

- Constant Folding.
- Expression Simplification.
- Operation Simplification (involves multiplies, divides, and mods by powers of two).
- Optimizing Expressions in a Logical Context (involves expressions which contain logical operators).
- Loop Construct Optimization.
- Switch Statement Optimization.
- Automatic Allocation of Register Variables.

The compiler may do many specific things for each type of optimization. The descriptions which follow contain examples to illustrate the kinds of things which are done for each type of optimization; they do not show every specific optimization performed by the compiler.

Note

In the general examples which follow, **E** represents any expression, **C** represents any constant, **!0** represents a constant with a non-zero value, and other operator symbols are their C equivalents.

Constant Folding

Whenever an expression contains operations made on constants, the compiler combines the constants to form a single constant. By folding constants, the compiler can eliminate the code which would otherwise be generated to perform the operations. A general and specific example of constant folding is shown below.

C1 * C2 - C3 / C4	⇒	C5
i = 4 * 3 - 10 / 2;	⇒	i = 7;

Constant Folding Across Expressions

The compiler will rearrange integer expressions to fold constants.

(E1 + C1) + (E2 + C2)	⇒	(E1 + E2) + (C1 + C2)
(E1 * C1) * (E2 * C2)	⇒	(E1 * E2) * (C1 * C2)
(E1 + C1) * C2	⇒	(E1 * C2) + (C1 * C2)
(E1 << C1) * (E2 * C2)	⇒	(E1 * E2) * $((2^{C1}) * C2)$
i = (x * 3 + 1) * 3 + 2	⇒	i = x * 9 + 5

Maintaining Order of Evaluation

Parentheses force grouping (prevent constant folding) of floating-point expressions. The unary plus (+) operator may be used to force grouping of arithmetic expressions. The unary plus operator may not be used to force grouping of pointer expressions. For example:

i = x+4.141 + y+2.067 + 3.287;	⇒	i = x + y + 9.495;
i= x+4.141 + (y+2.067)+3.287;	⇒	i = x + +(y + 2.067) + 7.428;
i= x+4.141 + +(y+2.067)+3.287;	⇒	i = x + +(y + 2.067) + 7.428;

Expression Simplification

The compiler will simplify expressions, if possible, by using the basic laws, identities, and definitions of conditional, logical, bitwise, and arithmetic operations. Some examples of expressions which get simplified follow.

Conditional:

0 ? E1 : E2	⇒	E2
!0 ? E1 : E2	⇒	E1

Logical:

E && 0	⇒	0 (unless E has side effects; then E,0)
E || 0	⇒	E
E1 && !E2	⇒	!(!E1 || E2)

Bitwise:

E & 0	⇒	0 (unless E has side effects; then E,0)
E | 0	⇒	E
E ^ 0	⇒	E
E << 0	⇒	E

Arithmetic:

E + 0	⇒	E
-E1 - (-E2)	⇒	E2 - E1
E * 0	⇒	0 (unless E has side effects; then E,0)
E * 1	⇒	E
E / -1	⇒	-E
E % 1	⇒	0 (unless E has side effects; then E,0)

Operation Simplification

Multiplications (whether explicit or as a result of scaling an array index), divisions, and mods of integral types by constants which equal powers of two can be simplified to bitwise operations which are shorter and faster. Generally:

$E * (2^{C})$	⇒	E << C

E / (2^C)	⇒	E >> C
E % (2^C)	⇒	E & (2^C - 1)

Optimizing Expressions in a Logical Context

When expressions containing logical operators are used in a logical context (for example, to yield a "true" or "false" in a control flow statement test expression), the compiler will generate code which evaluates the expression piece by piece. For example, suppose the test expression for an **if** statement is two expressions ANDed together. The compiler generates code which evaluates the first expression and branches out if it is "false" (if, at run-time, the first expression is "false", the second expression will not be evaluated). The compiler also generates code to evaluate the second expression in case the first is "true". The code generated as a result of this optimization is smaller and faster. Several "pseudo code" examples of optimizations on expressions in a logical context are shown below.

if (0) goto label	⇒	(Nothing.)
if (!0) goto label	⇒	goto label

if (E1 || E2) goto label	⇒	if (E1) goto label
		if (E2) goto label

if (E1 && E2) goto label	⇒	if (!E1) goto skip
		if (E2) goto label
		skip:

Loop Construct Optimization

The compiler places the evaluation of a loop construct's test expression at the end of the loop to avoid the execution of a "goto" at each loop iteration. A "goto" is generated to branch to the test for the first iteration. However, if the compiler can determine that the loop will execute at least once, the "goto" can be optimized out. Whenever the test expression becomes "false", execution simply "falls through".

The loop construct optimization can be generally expressed as follows.

while (E) { statements }	⇒	goto end
		beginning:
		{ statements }
		end: if (E) goto beginning
for (i = 0; i < 10;)	⇒	i = 0
{ statements }		beginning:
		{ statements }
		if (i < 10) goto beginning

Switch Statement Optimization

If there is code associated with at least 25% of the cases in a switch statement, the compiler will generate a jump table to access the code associated with each case. If less than 25% of the cases have associated code, the compiler will generate a hybrid binary/linear search to access the cases. The linear search can be up to four items long, otherwise a binary test is performed.

Automatic Allocation of Register Variables

Operating on variables which reside in registers is faster and more efficient than operating on variables in memory. The C compiler will automatically allocate variables to registers even in the absence of the **register** storage class specifier. Note that the presence of the **auto** storage class specifier prevents this optimization. For more information on the algorithm used by the compiler to allocate these variables, see the "Register Usage" section in the "Compiler Generated Assembly Code" chapter.

String Coalescing

When the compiler finds identical string constants, it stores them at a single memory location. In the following example, both string1 and string2 will point to the same memory location containing the string "abcde":

```
char *string1, *string2;
string1 = "abcde";
string2 = "abcde";
```

Only string constants allocated by the compiler are coalesced. For example, the following strings will not be coalesced because the user, rather than the compiler, is allocating the storage:

```
char string3[8] = "abcde";
char string4[8] = "abcde";
```

Note Trying to change the value of a string constant may cause unwanted side effects.

The compiler treats string literals as constants. Do not attempt to change the contents of a string which has been defined as a string literal. Be especially careful if you are using character pointers. For example, the following statements will change the value of *both* string1 and string2 to "abXde":

```
char *string1, *string2;
string1 = "abcde";
string2 = "abcde";

*(string1 + 2) = ’X’;
```

The compiler will not warn you about this.

The Optimize Option

The "optimize" command line option (**-O**) causes the compiler to use a more exhaustive algorithm in an attempt to generate locally optimal code; it also causes the compiler to run the peephole assembly code optimizer (unless the "generate run-time error checking code" option is also specified, in which case the "optimize" command line option is ignored).

You may find it easier to debug your code if you do not use the "optimize" option. Optimizations may make it difficult to follow the program flow. After the code is executing properly, use optimization to improve execution speed or to shrink the size of the executable code.

Time vs. Space Optimization

By default, the **-O** option causes the generated code to be optimized for space. That is, the compiler tries to generate as few bytes of code as possible (even, occasionally, at the expense of execution speed). However, if optimizing for time is more important (in other words, the generated code should execute as fast as possible), you can append the "time" option to the "optimize" option (**-OT**). Optimizing for time will cause the compiler to use more space if machine cycles can be saved. The listings in figure 5-1 give an example of a time vs. space trade-off.

OPTIMIZED FOR SPACE (Default).

```
1   struct test {
2           int     a,b,c,d,e,f;
3   } x, y;
4
5   main()
6   {
                                XDEF    _main
                        _main
7           y = x;
 00000000 207C 0000 0018  R         MOVE.L  #_y+0,A0
 00000006 227C 0000 0000  R         MOVE.L  #_x+0,A1
 0000000C 7205                      MOVEQ   #6-1,D1
                        L0_StatAssign
 0000000E 20D9                      MOVE.L  (A1)+,(A0)+
 00000010 51C9 FFFC                 DBF     D1,L0_StatAssign
8   }
--------------------------------------------------------------------------------
```

OPTIMIZED FOR TIME. (More bytes used to accomplish structure assignment, but code executes faster.)

```
1   struct test {
2           int     a,b,c,d,e,f;
3   } x, y;
4
5   main()
6   {
                                        XDEF    _main
                                _main
7           y = x;
        00000000 207C 0000 0018  R         MOVE.L  #_y+0,A0
        00000006 227C 0000 0000  R         MOVE.L  #_x+0,A1
                                           REPT    6
                                           MOVE.L  (A1)+,(A0)+
                                           ENDR
        0000000C 20D9                      MOVE.L  (A1)+,(A0)+
        0000000E 20D9                      MOVE.L  (A1)+,(A0)+
        00000010 20D9                      MOVE.L  (A1)+,(A0)+
        00000012 20D9                      MOVE.L  (A1)+,(A0)+
        00000014 20D9                      MOVE.L  (A1)+,(A0)+
        00000016 20D9                      MOVE.L  (A1)+,(A0)+
8   }
```

Figure 5-1. Example of Time vs. Space Optimization

Multiplication Simplification

In addition to the powers-of-two optimizations as described in the preceding "Operation Simplification" section, multiplications by constants up to 1024 (whether explicit or as a result of scaling an array index) are optimized to sequences of shifts and adds or subtracts. Shifts and adds or subtracts are typically less time expensive but more space expensive than the corresponding multiplications (i.e., these optimizations occur more often in the presence of the "time" option to the "optimize" command line option). For example:

E * 10	⇒	((E << 2) + E) << 1
E * 29	⇒	(((E << 3) - E) <<2) + E

Maintaining Debug Code

The compiler normally generates code which makes the resulting programs easier to debug with an HP emulator or simulator. This debug code includes:

1 Generation of no-operation (NOP) instructions preceding all labels. This provides unique addresses for all labels. It also avoids certain problems associated with instruction prefetch.

2 Buffering of the frame pointer on the stack at function entry and restoration of the frame pointer at function exit, even when this is known to be unnecessary. (Every function begins with a LINK instruction and ends with an UNLINK instruction followed by a RTS or RTE.)

When the "optimize" option is specified, this debug code is optimized out. However, if you wish the compiler to generate debug code and perform the other optimizations, use the "generate debug code" option with the "optimize" option. See the on-line man pages for more information on the compiler command line options.

Peephole Optimization

The peephole optimizer, which is run when the "optimize" command line option is specified, adds another pass to the compilation process. The peephole optimizer examines the assembly language instructions generated by the compiler and performs the optimizations described in the following subsections.

Branch (Jump) Shortening

Perhaps the most common peephole optimization is branch shortening. Neither the compiler (by itself) nor the assembler is capable of determining the distance of a forward branch. Consequently, 32-bit PC relative branches are generated by default.

The peephole optimizer, on the other hand, is capable of determining the distance of forward branches, and it will replace long branch instructions with byte or word sized instructions wherever possible.

Tail Merging

When two blocks of code end in identical branches, the peephole optimizer checks if the blocks have the same tail (ending) statements. If the blocks do have identical tail statements, the peephole optimizer will replace the first tail with a "goto" the second. If this would cause an additional branch to be executed, it is not performed when "optimize for time" is specified. For example:

```
. . .                                 ⇒     . . .
{ tail 1 }                            ⇒     goto sametail
goto label                            ⇒     . . .
. . .                                 ⇒     sametail:
{ tail 2 }  (Same as tail 1.)         ⇒     { tail 2}
goto label                            ⇒     goto label
. . .                                 ⇒     . . .
label:                                ⇒     label:
```

Redundant Register Load Elimination

When the peephole optimizer detects that a register is being loaded with a value it already contains, the second load is eliminated. (Compare to "Strength Reduction" below.)

```
MOVE.L (S_i+0,A6),D0
. . .
MOVE.L (S_i+0,A6),D0                        ; This instruction is removed.
```

Redundant Jump Elimination

When one jump occurs immediately after another jump, the two jumps are combined to form a single jump. Note that this optimization is performed on the generated assembly code, but a C code equivalent example would be the following:

```
if (x == y) goto aaa;          ⇒     if (x == y) goto bbb;
. . .                                . . .
aaa:goto bbb                         aaa: goto bbb;
. . .                                . . .
bbb:                                 bbb:
```

Unreachable Code Elimination

As compilers normally generate code, they can produce assembly instructions which will never get executed. The peephole optimizer can recognize unreachable assembly instructions and remove them.

Strength Reduction

Strength reduction refers to optimizations which can be made due to the optimizer's ability to remember the contents of registers. For example, the compiler may generate code to move a variable into one register, and later generate code to move the same variable into another register. The peephole optimizer can replace the second move with a move from the first register to the second (which is shorter and faster). Two bytes will be saved by the example strength reduction optimization shown below.

```
MOVE.L  (S_i+0,A6),D0          ⇒     MOVE.L  (S_i+0,A6),D0
MOVE.L  (S_i+0,A6),D1          ⇒     MOVE.L  D0,D1
```

Destination/Source Swapping

When generated code operates on a working register to yield a result, and the result is moved to a register variable, the peephole optimizer can eliminate the last move by causing the operations to be made on the register variable in the first place. This optimization is called destination swapping; an example is shown below.

```
MOVE.L  (S_i,A6),D0          ⇒     MOVE.L  (S_i,A6),D2
ADD (S_x,A6),D0              ⇒     ADD (S_x,A6),D2
MOVE.L  D0,D2
```

Source swapping refers to a similar situation where the value of a register variable is moved to a working register before being used in a calculation. In this case, the peephole optimizer will recognize that the calculation may just as well be made using the register variable, and a MOVE instruction will be saved.

Redundant Test Removal

Sometimes the compiler will generate a TST instruction without knowing that the condition codes were set by a previous instruction. The peephole optimizer can recognize this situation and remove the redundant test. For example:

```
ADD.L  D0,(S_i,A6)
TST.L  (S_i,A6)                    ; This instruction is removed.
BNE   L_001
```

Register Variable Reallocation

If the peephole optimizer can determine that a working register (A0, A1, D0, D1, FP0 or FP1) is not used in a function, it will reallocate a register variable to the working register. This optimization will save a "push" instruction on function entry and a "pop" instruction on function exit.

Effect of *volatile* Data on Peephole Optimizations

Any function that includes a **volatile** declaration or which follows any **volatile** declaration in a file will not have "data motion" optimizations performed on it. Data motion optimizations include redundant load elimination, strength reduction optimizations, source and destination swaps, and redundant test removal.

These optimizations account for considerably less than half of the space savings and roughly half of the speed savings that the peephole optimizer is capable of.

Branch shortening and branch structure simplification optimizations (tail merging, redundant jump elimination, and unreachable code elimination) are unaffected by **volatile** data.

Function Entry and Exit

The **-O** option also affects function entry and exit code. Whenever a called function has no parameters, no automatics, and returns a result whose size is eight bytes or less, the LINK and UNLINK instructions which are used to push the old stack frame pointer at function entry and restore the frame pointer on exit are not generated.

In-Line Expansion of Standard Functions

Certain standard functions have traditionally been implemented as macros (e.g., **putc()**), and errors are generated if such functions are assigned to function pointers. All other standard functions in **libc** and **libm** have traditionally been implemented with calls to support library routines. This is consistent with C's notion of having no built-in functions, and the user could rewrite these routines at will.

With the advent of standard definitions for these functions defined by ANSI, it is possible to expand these functions in-line, generally saving both time and space. This is particularly true in the presence of the "generate code for the 68881/2" command line option when a routine such as **tan()** reduces to a single instruction.

Of course, the assignment of these routines to function pointers requires that they be called rather than expanded in-line.

Routines that are expanded in-line even without the "generate code for the 68881/2" command line option are:

```
strcmp    strcpy    strlen
fabs (68040 only)    sqrt (68040 only)
```

For the 68020 and 68030, routines that are expanded in-line only in the presence of the "generate code for the 68881/2" command line option are:

```
acos
asin
atan
cos
cosh
exp
fabs
fint
fetoxm1
flog2
flognp1
ftentox
ftwotox
fatanh
fsincos
log
log10
sqrt
sin
sinh
tan
tanh
```

The appropriate header file (**string.h**, **math.h**, or **m6888x.h**) must be included for in-line expansion to occur.

68332 TBL functions

For the 68332, the following functions are expanded in-line to give you direct access to the TBL instruction:

```
tableS        interpolateS
tableSN       interpolateSN
tableU        interpolateU
tableUN       interpolateUN
```

Preventing expansion

Because it is conceivable that someone might want to substitute their own function for a standard one, the "do not expand standard functions in-line" (**-Wc,-I**) command line option turns off this default optimization.

It is also possible to prevent in-line expansion by using your own header file. Comments in the supplied header file explain how the definitions affect optimization.

What to do when optimization causes problems

Occasionally, the peephole optimizer can make incorrect assumptions, resulting in code that does not execute properly. Use the **-Wo,-m** command-line option to eliminate some of the risky optimizations (especially common sub-expression optimizations). If the code still doesn’t execute properly, you may need to avoid the **-O** optimizations.

6

Embedded Systems Considerations

Issues to consider when using the C compiler to generate code for your target system.

Execution Environments

The compiler cannot know the design of your target system. Therefore, all high-level functions and library routines depend on environment-dependent libraries to supply low-level hooks into the target execution environment.

The environment-dependent routines which are supplied with the compiler allow programs produced by the compiler to execute in an emulator. The supplied routines also support the debugger/simulator. Use these files as examples to create your own environment-dependent routines. We *expect* that you will need to modify the supplied files. You must use your own knowledge of your target system to decide what changes must be made.

Monitor and mon_stub

Current HP emulators use background monitors instead of foreground monitors. The environment files do not have a monitor program. Instead, there is a program (and accompanying source) called **mon_stub**. The **mon_stub** program completes the environment in the absence of the emulator monitor program. **mon_stub** does the following:

- Contains the trap vector table.
- Declares identifiers to satisfy external references in the *env.a* environment-dependent library.
- Acts as a template when you create your own version of the environment-dependent routines.

Common problems when compiling for an emulator

If you plan to execute your program in an emulator environment, follow these guidelines:

- Copy emulation configuration files (***.EA**) from the environment directory to a local directory prior to using.
- Use **#pragma SECTION DATA=idata** to specify the section for "initialized" data external declarations when using the **-d** option (separate initialized and uninitialized data).

Loading supplied emulation configuration files

Symptoms: In the emulator, one of the two supplied emulation configuration files is loaded from the directory */usr/hp64000/env/hp<emulator_environment>* and the following error message appears:

```
ERROR:  Cannot build
     /usr/hp64000/env/hp<emulator_environment>/ioconfig
```

Description: There are two forms of emulator configuration files. The first form (**.EA**), which is supplied, is an ASCII file. The second form (**.EB**), which is created from the ASCII file by the emulator, is a binary file. This binary file is not portable between versions of HP 64000 emulators and therefore not supplied.

When loading a configuration file, the emulator attempts to create the binary version of the file if one does not already exist. This binary file is created in the same directory as the ASCII file. The directory which contains the supplied configuration files is not meant to be modified and is write-protected. In order to use the supplied configuration file, it must first be copied to a local (writable) directory.

Using the "-d" option

Symptoms: During compilation, *cc68k* displays the following warning:

```
warning- Extern ’variable_name’ assumed to be in UDATA.
```

Description: The "Separate Initialized and Uninitialized Data" option (**-d**) causes the compiler to place static variable definitions with initializers in section **idata** by default, and static variable definitions without initializers in section **udata** by default. When an external declaration of a static variable is encountered the compiler assumes the external variable is uninitialized, places the external declaration in section **udata**, and issues a warning regarding this assumption. It is very important that if the external is instead an initialized variable that this warning be heeded and the external declaration placed in the proper section (**idata**). To do this, place a **#pragma SECTION DATA=idata** directive before the initialized variable’s external declaration and a **#pragma SECTION UNDO** following it. The second pragma merely "undoes" the first pragma. See the "Embedded Systems" chapter for more details on using these pragmas.

Section Names

Section names are used by the linker/loader to locate program code and data at the addresses appropriate for the target system environment. Code generated by the compiler is placed in relocatable program sections as follows:

- Executable code is placed in the PROG section (named **prog** by default).
- Static variables are placed in the DATA section (named **data** by default).
- Constants and string literals are placed in the CONST section (named **const** by default).

The section name information is also used when specifying (through a compiler command option) the addressing modes to be used for symbol accesses from one section to another. (For more information on specifying addressing modes, refer to the "Addressing Modes" section which follows.)

The default names given to the PROG, DATA, and CONST sections (**prog**, **data**, and **const**, respectively) may be changed with the SECTION pragma in the C source file.

The linker checks that external functions or data declared to be in a particular section are indeed found in that section. This checking is defeated for the default sections named **prog**, **data**, and **const**. This is done to accommodate existing C programs which often declare **extern** C library functions without any SECTION pragma. Note that all header (.h) files for libraries do include the appropriate **extern** declarations with the correct section pragmas.

If there are multiple declarations for the same symbol within a single file, the compiler checks that the section in which the symbol is declared is the same in all cases; if it is not, an error is reported. An exception is made for this checking when both declarations are external references and the "specify addressing modes" (**-m**) command line option has not been given.

#pragma SECTION

Syntax:

```
#pragma SECTION [PROG=pname] [DATA=dname] [CONST=cname]
#pragma SECTION [PROG=address [BP]] [DATA=address [BP]] [CONST=address [BP]]
#pragma SECTION [PROG=pname] [UDATA=udname] [IDATA=idname] [CONST=cname]
```

```
#pragma SECTION [PROG=address [BP]] [UDATA=address [BP]] [IDATA=address [BP]]
[CONST=address [BP]]
#pragma SECTION UNDO
```

Description

The first form of this pragma causes the program, static data, and static constant information to be placed in sections named *pname*, *dname*, and *cname* respectively until the next **SECTION** pragma is encountered. The linker also expects to find external functions and data in these named sections.

In the second form, absolute addresses are given in place of the segment names causing the subsequent information to be ORG’d starting at the given address. An optional **BP** may be given after such an address to indicate that it is in the base page and that absolute short addressing should be used to access that location rather than absolute long.

Note

When absolute addresses are used, all information (program, data, or constant) to be ORG’d must immediately follow the **#pragma SECTION** line and come prior to any information (program, data, or constant) which is output in another named or ORG’d segment. For example:

```
#pragma SECTION DATA=0x1000
int i, j, k;
const int l;
int m, n, o;
```

will cause an error since constant integer "l" is output in another section (const) and since integers "m, n, o" also need to be ORG’d as they are data. Corrected this becomes:

```
#pragma SECTION DATA=0x1000
int i, j, k;
int m, n, o;
const int l;
```

Other cases that cause information to be put out in new sections include: **extern** definitions and string literals.

The third and fourth forms listed are the same as the first two forms with **UDATA** and **IDATA** substituted for **DATA**. These forms make sense only in the presence of the "separate initialized and uninitialized data" option (**-d**) which forces separation of explicitly initialized data from implicitly initialized (or uninitialized with **-u**) data. Non-constant static data items explicitly initialized by means of a C initializer go into the **IDATA** named section. Non-constant static data items not explicitly initialized by means of a C initializer go into the **UDATA** named section.

Watch for the "extern variable assumed to be in UDATA" warning message. If the variable is initialized, place it in the **IDATA** section by naming the **DATA** section to be the same as the **IDATA** section. For example:

```
#pragma SECTION UDATA=UDataSec IDATA=IDataSec
extern int x;                      /*                        */
#pragma SECTION UNDO               /*  Both x and y will go  */
                                   /*  in the UDATA section. */
#pragma SECTION IDATA=IDataSec     /*                        */
extern int y;                      /*                        */
#pragma SECTION UNDO

#pragma SECTION DATA=IDataSec      /*  z will go in the      */
extern int z;                      /*  IDATA section.        */
#pragma SECTION UNDO
```

The absolute addresses and segment names may be intermixed for the three (four, counting **UDATA** and **IDATA**) different information types (program, static data, static constant) in the same **SECTION** pragma. If the target section is not specified for one of the information types, then it remains unchanged.

The last form, **#pragma SECTION UNDO**, "undoes" the effect of the immediately preceding **SECTION** directive. That is, it restores the name (or address) of any section renamed (or ORG'd) in the last directive. This form is useful at the end of **#include** files to restore the section environment which existed prior to the **#include** file. (Include files must contain **SECTION** directives to define the sections that externs are in.)

The SECTION pragma must be placed outside a function body.

Note

#pragma SECTION UNDO is implemented by a one-level-deep stack. That is, only the most recent **SECTION** pragma may be "undone" or, said another way, two **#pragma SECTION UNDO**s in a row will not undo two **SECTION** pragmas. This is of particular importance when an include file further includes other files. Since include files will generally surround their **extern** declarations with a **SECTION-SECTION UNDO** pair, care must be taken not to put an include inside of this pair as it will result logically in two "UNDO"s in a row.

Addressing Modes

The C compiler allows you to specify the addressing mode to be used when references are made from one section to static variables and functions defined in the same or other sections. Only references to variables and function calls are affected; branches are always PC relative. Sections are the named sections which are understood by the linker.

In some situations, selecting the appropriate addressing mode is critical. In other situations, selecting the appropriate addressing mode (short vs. long) can be a way of generating more efficient code.

The appropriate addressing mode to use depends on such questions as:

- **Is the data on base page?** (The absolute short addressing mode can be used to access this data; instructions will use 16-bit addresses instead of 32-bit addresses.)
- **Is the data near to the accessing code?** (The PC relative short addressing mode can be used to read locations within +/- 32K bytes; instructions will use 16-bit signed displacements instead of 32-bit absolute addresses. Program Counter relative writes are not directly supported by the 68000; they can be simulated by the compiler, but they will cost more time and space than absolute long writes.)
- **If the data is not on the base page, is it less than 64K bytes in length?** (The A5 relative short addressing mode can be used to access data in a 64K byte block; instructions will access data with 16-bit signed displacements instead of 32-bit absolute addresses. The effect is a second base page.)
- **Does the code need to be position independent?** (No absolute addresses may be used.)
- **Does the code operate on run-time dynamically relocated data?** (A5 relative addressing modes must be used.)

There are six 68000 addressing modes available to the C programmer: absolute short, absolute long, PC relative short, PC relative long, A5 relative short, and A5 relative long. The default addressing mode used is absolute long.

Specifying addressing modes

Addressing modes are specified with a command line option. You can specify that a certain addressing mode be used on accesses from one section to another, from one section to all sections, from all sections to one section, or from all sections to all other sections. Sections are named by default (**prog**, **data**, and **const**) or with the SECTION pragma in the C source file. (See the on-line man page for a description of the "mode" command line option. See also the section on "Addressing Modes Used in Libraries" in the "Libraries" chapter.)

Note

The 68000 does not support the PC relative long and A5 relative long addressing modes directly. The compiler generates code to calculate the proper addresses. These instructions are more expensive in terms of time and space than absolute long mode instructions.

When to use certain addressing modes

Certain situations are associated with the use of a particular addressing mode. Listed below are the situations generally associated with the six addressing modes.

Mode	Situation
Absolute Short	The absolute short addressing mode can be used when accesses are made to code or data in the address range 0xFFFF8000 to 0x7FFF (base page).
Absolute Long	This is the default addressing mode. The absolute long addressing mode is typically used when accesses are made to locations outside the base page.
PC Relative Short	The program counter relative short addressing mode is used when you wish to create position independent code and when that code accesses locations within +/- 32K bytes of the current program counter location.
PC Relative Long	The program counter relative long addressing mode is used when you wish to create position independent code and when that code accesses locations which are greater than +/- 32K bytes away from the current program counter location.

A5 Relative Short

The A5 relative short addressing mode is used when accessing locations relative to an address (in register A5) which are less than +/- 32K bytes away from that address. The run-time value of the A5 register can be specified when linking, and register A5 can be initialized with that run-time value (as it is in the setup object file **crt0**).

A5 Relative Long

The A5 relative long addressing mode is used when accessing locations relative to an address (in register A5) which are greater than +/- 32K bytes away from that address. The run-time value of the A5 register can be specified when linking, and register A5 can be initialized with that run-time value (as it is in the setup object file **crt0**).

Short vs. long

The "short" addressing modes are associated with 16-bit offsets. In the case of the absolute mode, short means that addresses can be accessed with 16-bits; in other words, they are located in the address range -32K to +32K (0xFFFF8000 to 0x7FFF). In the case of the PC and A5 relative addressing modes, short means 16-bit signed displacements are used. Signed 16-bit displacements allow accesses to locations within +/- 32K bytes of the program counter or address in A5.

Note

When using the short addressing modes, the compiler will assume that program or data references are within the short (signed 16-bit) range. If the accessed code or data does not fit within that range, assembler or linker errors will result.

The "long" addressing modes are associated with 32-bit quantities. The absolute long mode, which is the default, allows locations anywhere in the microprocessor address space to be accessed.

In the PC and A5 relative addressing modes, long also means that locations anywhere in the microprocessor address space can be accessed, except that these modes use 32-bit signed offsets.

Using the short addressing modes generates more efficient code in the sense that instructions will have one less word of extension, causing shorter code which executes faster.

For the 68000, note that the PC and A5 relative long modes are not supported by the 68000, and the compiler generates code to calculate the proper addresses. Consequently, these modes are less efficient.

Absolute addressing modes

Since a section is the smallest piece of code for which an addressing mode can be selected (with the exception of using assembly language), the absolute addressing mode you choose (long or short) will depend upon where sections are to be located with the linker. For example, suppose one of your programs calls a library function; if the library function's program section is to be located in the base page, you would then specify the absolute short mode for references from your program section to the library program section (**lib**, **libc**, **libm**, etc.).

PC relative addressing modes

The PC relative addressing modes allow you to create position independent code. Also, in certain situations, the PC relative short addressing mode allows you to access data or call functions more efficiently than the absolute long addressing mode.

Position independent code

In general, a position independent code section uses PC relative function calls and branches and accesses only local data (on the stack). This is readily accomplished by specifying PC relative short (if the segment fits into 64K bytes) or long as the addressing mode used for accesses from the position independent section to itself.

If the position independent code section must declare or access static data, different addressing modes will be appropriate depending on the desired functionality. First of all, it may be acceptable to have the code depend on the location of static data (i.e., absolute addresses may be used). Secondly, it may be desirable to force the static data to be located in a fixed location *relative to* the position independent code and, thus, PC relative access would be appropriate. Finally, it may be necessary to allow the code to be located anywhere and its static data to be located anywhere. In this case, A5 relative addressing should be used, and A5 should be loaded with the appropriate value relative to the location of the static data. In these last two cases (static data being accessed PC relative or A5 relative), care should be taken to avoid absolute addresses of static data inadvertently appearing as load-time initializers. For example:

```
int j;
int *i = &j;
char *p = "abc";
```

The declarations above would result in "i" being initialized with the absolute address of "j" and in "p" being initialized with the absolute address of the constant string "abc".

Accessing near locations

The PC relative short addressing mode can be used to advantage when a program accesses code or data which is nearby. If locations are within +/- 32K bytes of the current program counter, they can be accessed with PC relative short instructions which use 16-bit signed displacements. Alternatively, absolute long mode instructions would use 32-bit absolute addresses.

Note

The 68000 does not support PC relative data writes. The compiler creates this addressing mode from multiple instructions. These instructions are more expensive in terms of time and space than absolute long mode instructions.

Branches within functions (68000 only)

By default, the compiler generates PC relative short branches for "goto"s in control flow constructs within functions. Rarely, the code around which a "goto" must jump will exceed 32K bytes. In this case, an assembly time error will result.

The "big switch tables" option the the C compiler causes the compiler to use PC relative long branches. Such branches are not directly available in the 68000 instruction set and are "manufactured" from a series of instructions. The optimizer will shorten these except when required.

A5 relative addressing modes

The A5 relative addressing modes can be used to advantage in many types of situations. A few of the most general situations are described here.

Creating a second "base page"

Suppose that a program accesses a 64K byte block of data located at an address off of the base page. Without A5 relative addressing, this data would have to be accessed with an absolute long or PC relative addressing mode. However, with A5 relative addressing, you can load register A5 with the address of the mid-point of the data section and access 64K bytes of data using the A5 relative short mode. The A5 relative short addressing mode is more efficient than the absolute long mode because it uses 16-bit signed displacements instead of 32-bit absolute addresses.

If the "run-time" value of A5 is set equal to the base address of the data section plus 8000H, then 64K bytes of data may be accessed with 16-bit displacements using the A5 relative short addressing mode.

When using the A5 relative short addressing mode in this way, instructions such as "move memory (relative to A5) into register" generate 2 words of code instead of the 3 words of code generated by instructions such as "move memory into register."

Using the A5 relative short addressing mode in this manner can be thought of as creating a second "base page."

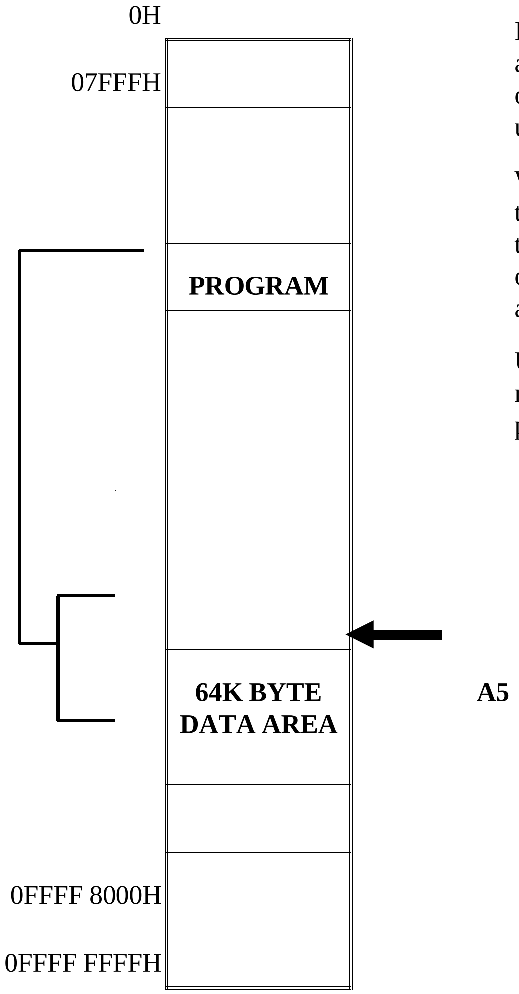


Figure 6-1. Creating a Second Base Page

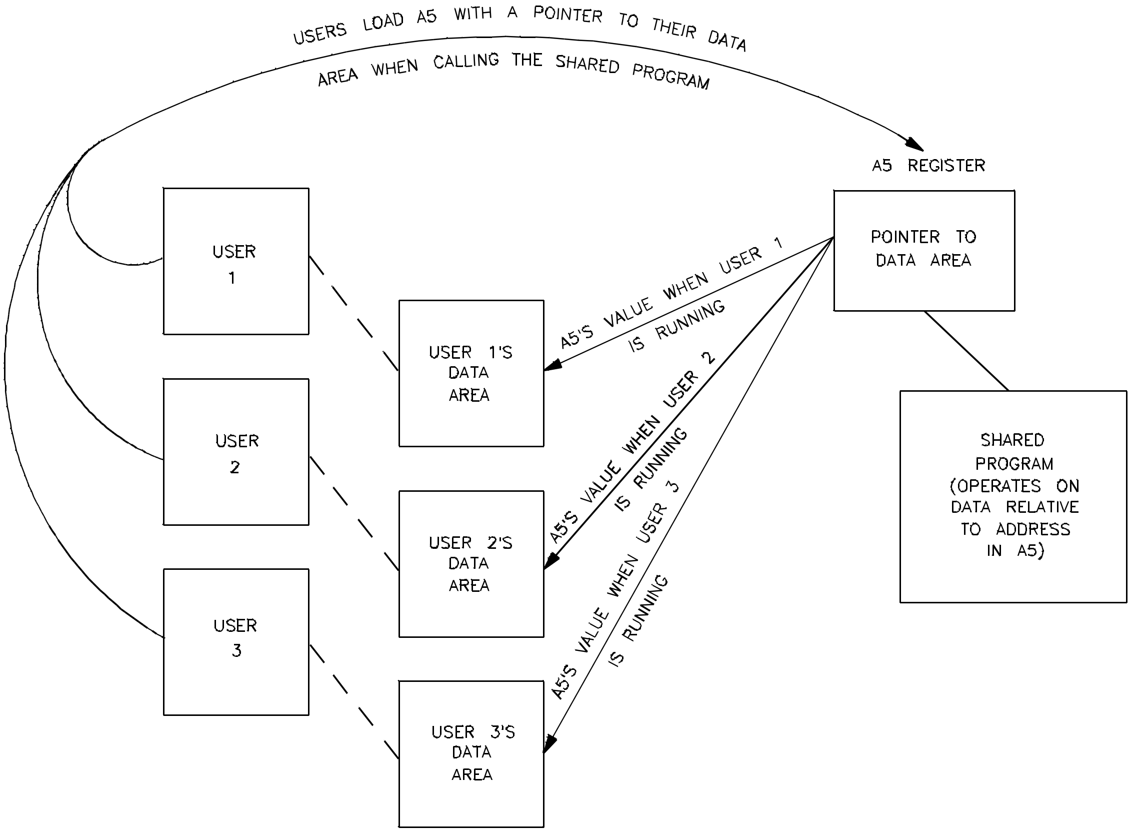

USERS LOAD A5 WITH A POINTER TO THEIR DATA
AREA WHEN CALLING THE SHARED PROGRAM
A5 REGISTER
POINTER TO DATA AREA
USER 1
USER 2
USER 3
USER 1'S DATA AREA
USER 2'S DATA AREA
USER 3'S DATA AREA
A5'S VALUE WHEN USER 1 IS RUNNING
A5'S VALUE WHEN USER 2 IS RUNNING
A5'S VALUE WHEN USER 3 IS RUNNING
SHARED PROGRAM (OPERATES ON DATA RELATIVE TO ADDRESS IN A5)

Figure 6-2. Shared Programs

Shared programs

When a program is shared, it may have to operate on separate data areas associated with each user. Only local variables and ROM variables (global or static) may be used by shared programs. When programs are shared, the A5 register is used as a pointer to the user’s data area.

Other addressing mode considerations

When embedding assembly language in C source files, make sure that your assembly language instructions agree with the addressing mode you have selected for that section. For example, you would not code absolute mode jump instructions in a PC relative mode program section.

RAM and ROM Considerations

This section addresses special considerations of loading your programs into RAM and ROM environments.

Initialized data

The C language specifies that, without explicit initialization, external and static variables will be initialized to zero. Declarator initializers allow you to specify initial values other than zero.

These initial values, or default values of zero, are written to static variables at load-time. Programs executed in operating systems, in emulation environments, or in simulation environments, have a "load-time" and initialization is possible.

Embedded environments, however, have no "load-time", and statics and externals cannot be initialized (either to zero or any other value). As an example, when a target system is powered up, the contents of RAM data locations are not known.

Symbols declared with the **const** type modifier are considered to be ROM locations and are initialized by definition.

The "uninitialized" option

There is an "uninitialized data" option (**-u**) to the compiler which will cause warning messages to be printed whenever static initializers are used in non-constant declarations. Also, the generated assembly language declarations no longer initialize static data to zero (as is done when the "uninitialized data" option is not specified).

This option cannot check for the use of a static or external variable which has not been assigned a value (although the compiler generates warnings occasionally), so make sure your programs do not assume an initialized value.

Where to load constants

For ROM/RAM embedded systems, the program and constants will ultimately reside in ROM. In anticipation of this environment, the default sections **prog** and **const** contain ROMable information, and the default section **data** contains RAMable information.

Embedded Systems with Mass Storage

Systems which load programs from mass storage differ from pure ROM/RAM systems in that a load time exists when alterable data can be initialized. The C language anticipates such a load time by allowing variable data to be declared with initializers. When static data is declared with an initializer, such initialization occurs at load time. Indeed C specifies that static data which is not explicitly initialized will be initialized to zero at load time.

To facilitate load time initialization of static data, a command line option has been provided to separate explicitly initialized data from uninitialized (or initialized to zero at load time) data into different named sections. By default, these sections are named *idata* and *udata*; but these names can be changed using **#pragma SECTION** (see above).

The value of the "separate initialized and unitialized data" option is that it allows the loader to load initialized static data *contiguously* into RAM from the *idata* section. It can then, if desired, initialize the *udata* section’s locations to zero in an efficient contiguous manner.

The use of the "separate initialized from uninitialized" option together with the "uninitialized data" option (described above) supports emulation of an environment with a load time (for initializing explicitly initialized static data) which does not initialize uninitialized data to zero. When used together, the compiler does not warn explicit initializations of non-constant static data, but places such data in section *idata* (by default). Static data which is not explicitly initialized, is reserved space in section *udata* (by default), but is not initialized to zero at emulation/simulation load time.

The "volatile" Type Modifier

The **volatile** type modifier is used in declarations to specify that an object’s value may change in ways unknown to the compiler. A **volatile** type modifier makes the compiler access an object literally, as specified in C statements. Literal interpretations of C statements can be important in programs which are closely tied to hardware such as memory mapped I/O devices or device drivers. The **volatile** type modifier is necessary because optimizations can take short-cuts, using methods which differ from the literal interpretation but which yield the same result.

The listings shown in figure 6-3 give an example of the effect given by the **volatile** type modifier. The top listing shows code in which the assignment of "io_port" to "secondValue" has been optimized into a "MOVE.L D0,D4" instruction which does not actually read "io_port" (whose value may have changed since its assignment to "firstValue"). The bottom listing shows the "io_port" variable declared with the **volatile** type modifier. Notice that the assignment of "io_port" to "secondValue" does not get optimized.

For the user who wants a controlled way of toggling an address line, it is guaranteed that a simple assignment to a **volatile** variable which has a size equal to the data bus width of the target processor will cause exactly one write. An access of such a variable will cause exactly one read. For example:

```
volatile int    *p = (int *)1234;          /* int size = bus width.
                                              1234 is address of I/O port.  */
main()
{
        *p = 0;        /* Exactly one write to address 1234.  */
        *p;            /* Exactly one read of address 1234.   */
}
```

A pointer-to-**volatile** cannot be assigned to a pointer-to-non-**volatile** without a cast.

Note

If the "word align data" option is on, **short** and **int** variables may be accessed with *two* reads or writes instead of just one.

```
 1   int io_port;
 2
 3   main()
 4   {
                                     XDEF    _main
                             _main
 00000000 2F04                       MOVE.L  D4,-(A7)
                             *
                             * Register ’D0’ is register variable ’S_firstValue’.
                             *
                             * Register ’D4’ is register variable ’S_secondValue’.
                             *
                             * Register ’D1’ is register variable ’S_tmp’.
                             *
 5           int firstValue, secondValue, tmp;
 6
 7           firstValue = io_port;
 00000002 2039 0000 0000  R         MOVE.L  (_io_port+0).L,D0
 8           secondValue = io_port;
 00000008 2800                       MOVE.L  D0,D4
 9           tmp = firstValue;
 0000000A 2200                       MOVE.L  D0,D1
10   }
```

OPTIMIZATION PERFORMED

```
--------------------------------------------------------------------------------

 1   volatile int io_port;
 2
 3   main()
 4   {
                                     XDEF    _main
                             _main
 00000000 2F04                       MOVE.L  D4,-(A7)
                             *
                             * Register ’D0’ is register variable ’S_firstValue’.
                             *
                             * Register ’D4’ is register variable ’S_secondValue’.
                             *
                             * Register ’D1’ is register variable ’S_tmp’.
                             *
 5           int firstValue, secondValue, tmp;
 6
 7           firstValue = io_port;
 00000002 2039 0000 0000  R         MOVE.L  (_io_port+0).L,D0
 8           secondValue = io_port;
 00000008 2839 0000 0000  R         MOVE.L  (_io_port+0).L,D4
 9           tmp = firstValue;
 0000000E 2200                       MOVE.L  D0,D1
10   }
```

NO OPTIMIZATION PERFORMED

Figure 6-3. "volatile" Type Modifier Example

Reentrant Code

Reentrant code is code that can be interrupted during its execution and re-invoked by subsequent calls any number of times. A nonreentrant routine might, for example, operate on static data or external variables; if this routine is interrupted and called from somewhere else, the data it was originally operating on might be destroyed. Interrupt handlers and other routines which may be interrupted and called again must be reentrant.

The C compiler generates reentrant code.

Nonreentrant library routines

Most of the library routines which have been shipped with the compiler are reentrant. However, some of the libraries are not reentrant; they are listed below.

Table 6-1. Nonreentrant Library Routines

assert
atexit
calloc
close
fclose
fflush
fgetc
fgetpos
fgets
fopen
fprintf
fputc
fputs
fread
free
freopen
fscanf
fseek
fsetpos
ftell
fwrite
getc
getchar
gets
lseek
malloc
open
printf
putc
putchar
puts
rand
read
realloc
remove
rewind
scanf
setbuf
setvbuf
srand
strtok
strtol
ungetc
unlink
vfprintf
vprintf
write

Nonreentrant routines should not be called from interrupt handlers or other reentrant routines.

Some libraries use the global symbol *errno*. Note that the value of *errno* can be overwritten in a multitasking or reentrant environment.

Implementing Functions as Interrupt Routines

Interrupt routines are not intended to return values. Therefore, the type specifier **void** must be used to declare functions which you wish to implement as interrupt routines. The **INTERRUPT** pragma is used to specify that a function should be implemented as an interrupt routine.

#pragma INTERRUPT

This pragma specifies that the next encountered function be implemented as an interrupt routine. This means that all working registers are saved at function entry (in addition to the register variables which ordinarily are saved), no parameter passing or returned result is allowed, and a return from interrupt is generated at the return point. Note that only the next encountered function is affected**--**not subsequent functions.

The **INTERRUPT** pragma may be used any place a C external declaration may. An example of a function implemented as an interrupt routine is shown below.

```
#pragma INTERRUPT

void int_routine()
{
        .
        .
        .
}
```

Loading the vector address

Using the **INTERRUPT** pragma will cause all registers to be pushed onto the stack upon function entry, and a return from interrupt instruction is generated for function exit. However, you must make sure that the address of the function is loaded into the vector table. For example, the emulator monitor stub program uses some interrupt vectors. The source for **mon_stub.s** contains vector tables which may be modified to contain the address of your interrupt handler written in C.

In your own target system, it will be easiest to implement your vector table in C. For example, if you had implemented one routine totally in assembly language and named it "_asm_int_routine", you could declare your vector table and initialize it with:

```
extern void     asm_int_routine();

#pragma SECTION DATA=0x00

void (*vectorTable[])() = {.  .  ., asm_int_routine, .  .  .,
                           int_routine, .  .  .};

#pragma SECTION UNDO
#pragma INTERRUPT

void int_routine() {
        .
        .
        .
}
```

Eliminating I/O

Your embedded system may well have no file I/O capability. If this is the case, you can specify a linker command file which avoids the overhead of initializing emulation simulated I/O buffers for *stdin*, *stdout*, and *stderr*. See the description of cc68k in the on-line man page.

7

Libraries

Descriptions of the run-time and support libraries.

Two varieties of libraries are provided with the Motorola 68000 Family C Cross Compiler. First are the run-time libraries which contain routines required to do real number arithmetic, initializations, run-time debug checks, etc. Second are the support libraries for which both a "**.h**" include file and the library object code are provided.

Addressing Modes Used in Libraries

Three versions of the library routines are provided: position dependent versions, and two position independent versions.

The position dependent versions of the library routines access static data with the absolute long mode, call other library routines in the same named library section with the PC relative short mode, and call library routines across named library sections with the absolute long mode.

The position independent versions of the library routines access static data with the A5 relative long mode, call other library routines in the same named library section with the PC relative short mode, and call library routines across named library sections with the PC relative long mode.

The program-counter relative versions of the library routines access static data with the PC relative long mode, call other library routines in the same named library section with the PC relative short mode, and call library routines across named library sections with the PC relative long mode.

Library Names

The absolute and position-independent versions of the default libraries (which are included by cc68k when linking) are:

Table 7-1. Absolute/Position-Independent Library Names

Library:	Absolute Version (PROG Section Name)	Position-Independent Version (PROG Section Name)	PC-Relative Version (PROG Section Name)
Run-time library	lib.a (lib)	libpi.a (lib)	libpc.a (lib)
Support library	libc.a (libc)	libcpi.a (libc)	libcpc.a (libc)
Math library	libm.a (libm)	libmpi.a (libm)	libmpc.a (libm)

The run-time, support, and math libraries are included by the default linker command file. When the "generate code for the 68881/2" command line option is used, the 68881/2 run-time and math libraries are also included by the default linker command file. Because there are no real numbers in the support libraries, there is no 68881/2 version of the support library.

Controlling the Addressing Mode of the Calling Code

You can control the addressing modes used for calls to libraries in the same manner as you would control the addressing modes throughout your program. The "mode" option to the cc68k command allows you to specify the addressing modes which are generated when one section makes references to static variables or functions which are defined in the same or a different section.

Run-time library modules are all located in linker section name **lib**. The same addressing mode must be used to call run-time library modules throughout a source file.

Support and math libraries are located in section names **libc** and **libm**, respectively. The 68881/2 math library modules are located in section name **libm**. These section names may be used just as any other section names would be (for example, in **SECTION** pragmas or in the "mode" command line option provided include files are used). See the on-line man pages for a complete description of the cc68k command syntax and options.

Run-Time Library Routines

The run-time library, **lib.a** or **lib881.a**, contains routines used at run-time by the compiler-generated code to implement operations which, for one reason or another, are better accomplished in a subroutine than in-line. The reasons for encoding an operation in a run-time library routine instead of in-line vary from conserving space to minimizing repetition of in-line code to maintenance considerations (the same reasons functions are used in C).

Although run-time library routines are usually used only by the compiler, they may be called from assembly code (including embedded assembly code within the C source). Also, it is possible to replace any or all of the routines with your own routines. See the "Run-Time Library Description" chapter for descriptions of the interface and functionality of all run-time library routines.

Run-time libraries, unlike user routines, generally receive their parameters in the compiler’s "working" registers (A0, A1, D0, D1). Their results are generally returned in a similar manner as results which are returned from user routines. See the "Run-Time Library Description" chapter for detailed information on specific routines.

In the presence of the "generate code for the 68881/2" option, most run-time library routines are not called, but rather in-line 68881/2 instructions are used to perform the operation.

Support Library and Math Library Routines

In general, the implementation of the support library routines is likely to deviate subtly from the standard due to environment dependencies. Where possible, the sources for these environment-dependent routines (which are customized to HP development environments) are provided as part of the compiler product (see the chapters describing "Environment Dependent Routines").

Especially in the presence of the "generate code for the 68881/2" option, certain support and math library functions may be expanded in-line (see the "In-Line Expansion of Standard Functions" section in the "Optimizations" chapter).

Certain support and math library functions may be expanded in-line (see the "In-Line Expansion of Standard Functions" section in the "Optimizations" chapter).

Library Routines Not Provided

Several "standard" C library routines are **not** provided with the C compiler.

- **General Utilities**. The <stdlib.h> functions **abort**, **getenv**, and **system** are not supported.
- **Input/Output**. The <stdio.h> definitions **L_tmpnam**, **FILENAME_MAX**, and **TMP_MAX**, as well as the **rename**, **tmpfile**, and **tmpnam** routines, are not supported.
- **Signal Handling**. The <signal.h> routines are not provided because of their extreme environment dependencies.
- **Date and Time**. The <time.h> routines are not provided because of their extreme environment dependencies.

Include (Header) Files

The following is a list of include files which are shipped with the compiler:

assert.h Defines the macro **assert**.

ctype.h Defines the "character classification" macros (e.g., **isalnum**, **isalpha**, etc.).

errno.h Declares **errno** and macros used to test errno.

float.h Describes the IEEE single**-** and double-precision floating-point representations and contains definitions of the limiting values of floating-point types.

fp_control.h Declares the floating-point error functions. This header file also defines the macros which can be used as arguments to the **_set_fp_control** function, or to check the return value of the **_get_fp_status** function.

limits.h Contains definitions of the limiting values for integral types.

locale.h Declares the **setlocale** and **localeconv** functions and defines the **lconv** structure. Also defines the categories which the functions can change.

math.h Declares the standard math library routines and **HUGE_VAL**.

memory.h Declares **sbrk** and **_getmem**.

m68332.h Declares the **table** and **interpolate** routines.

m6888x.h Declares floating-point functions so that the 68881/2 FPU will be used.

setjmp.h Defines the **jmp_buf** type and declares the **setjmp** and **longjmp** functions.

simio.h Declares the simulated I/O functions and companion macros.

stdarg.h Provides the **va_list** type and the macros which are used to access variable-length argument lists, **va_start**, **va_arg**, and **va_end**. For a description of the variable argument list macros, see the entry for "va_list" in this chapter.

stddef.h Defines the **ptrdiff_t**, **size_t**, and **wchar_t** types and the **NULL** null pointer constant. This header file also defines the **offsetof** macro.

stdio.h Declares all the functions that handle input and output. This header file also defines the **FILE** type, buffering macros, file positioning macros, the maximum number of open files, and buffer size macros.

stdlib.h Defines the types **div_t** and **ldiv_t**, and also the macros **EXIT_SUCCESS**, **EXIT_FAILURE**, **RAND_MAX**, and **MB_CUR_MAX**. This header file also declares standard library functions.

string.h Declares the character string and memory operations.

List of All Library Routines

The following table lists all of the library routines shipped with this compiler.

An asterisk (*) in the **Index** column means that you can find a description of the routine in this manual by looking in the index.

The routines not marked with an asterisk are not described in this manual. These routines are run-time routines or subroutines used by the libraries. You should not use these undocumented routines in your programs because they are likely to be changed or even deleted in future versions of the compiler.

Index	Definition name	Library
	___tableS	libc.a
	___tableSN	libc.a
	___tableU	libc.a
	___tableUN	libc.a
	__assert	libc.a
	__bufsync	libc.a
*	__clear_fp_status	lib.a libm.a
	__dbl_to_str	libc.a
	__display_message	lib.a
	__doprnt	libc.a
	__doscan	libc.a
	__exec_funcs	libc.a
*	__exit	libc.a
	__filbuf	libc.a
	__findbuf	libc.a
	__findiop	libc.a
	__flsbuf	libc.a
*	__fp_control	lib.a libm.a
*	__fp_error	lib.a
*	__fp_error	libm.a
	__fp_errorf	lib.a
	__fp_errori	lib.a
*	__fp_status	lib.a libm.a
*	__get_fp_control	lib.a libm.a
*	__get_fp_status	lib.a libm.a

Index	Definition name	Library
*	__getmem	libc.a
	__memccpy	libc.a
	__readFile	libc.a
	__readStr	libc.a
*	__set_fp_control	lib.a libm.a
	__swrite	libc.a
	__wrtchk	libc.a
	__xflsbuf	libc.a
*	_abs	libc.a
*	_acos	libm.a
*	_asin	libm.a
*	_atan	libm.a
*	_atexit	libc.a
*	_atof	libc.a
*	_atoi	libc.a
*	_atol	libc.a
*	_bsearch	libc.a
*	_calloc	libc.a
*	_ceil	libm.a
*	_clearerr	libc.a

Index	Definition name	Library
*	_close	libc.a
*	_cos	libm.a
*	_cosh	libm.a
*	_div	libc.a
*	_errno	lib.a libc.a libm.a
*	_exp	libm.a
*	_fabs	libm.a
*	_fclose	libc.a
*	_feof	libc.a
*	_ferror	libc.a
*	_fflush	libc.a
*	_fgetc	libc.a
*	_fgetpos	libc.a
*	_fgets	libc.a
*	_floor	libm.a
*	_fmod	libm.a
*	_fopen	libc.a
*	_fprintf	libc.a
*	_fputc	libc.a

Index	Definition name	Library
*	_fputs	libc.a
*	_fread	libc.a
*	_free	libc.a
*	_frem	libm.a
*	_freopen	libc.a
*	_frexp	libm.a
*	_fscanf	libc.a
*	_fseek	libc.a
*	_fsetpos	libc.a
*	_ftell	libc.a
*	_fwrite	libc.a
*	_getc	libc.a
*	_getchar	libc.a
*	_gets	libc.a
*	_interpolateS	libc.a
*	_interpolateSN	libc.a
*	_interpolateU	libc.a
*	_interpolateUN	libc.a
*	_isalnum	libc.a
*	_isalpha	libc.a

Index	Definition name	Library
*	_iscntrl	libc.a
*	_isdigit	libc.a
*	_isgraph	libc.a
*	_islower	libc.a
*	_isprint	libc.a
*	_ispunct	libc.a
*	_isspace	libc.a
*	_isupper	libc.a
*	_isxdigit	libc.a
*	_labs	libc.a
*	_ldexp	libm.a
*	_ldiv	libc.a
*	_localeconv	libc.a
*	_log	libm.a
*	_longjmp	libc.a
*	_lseek	libc.a
*	_malloc	libc.a
*	_mblen	libc.a
*	_mbstowcs	libc.a
*	_mbtowc	libc.a

Index	Definition name	Library
*	_memchr	libc.a
*	_memcmp	libc.a
*	_memcpy	libc.a
*	_memmove	libc.a
*	_memset	libc.a
*	_modf	libm.a
*	_open	libc.a
*	_perror	libc.a
*	_pow	libm.a
*	_printf	libc.a
*	_putc	libc.a
*	_putchar	libc.a
*	_puts	libc.a
*	_qsort	libc.a
*	_rand	libc.a
*	_read	libc.a
*	_realloc	libc.a
*	_remove	libc.a
*	_rewind	libc.a
*	_scanf	libc.a

Index	Definition name	Library
*	_setbuf	libc.a
*	_setjmp	libc.a
*	_setlocale	libc.a
*	_setvbuf	libc.a
*	_sin	libm.a
*	_sinh	libm.a
*	_sprintf	libc.a
*	_sqrt	libm.a
*	_srand	libc.a
*	_sscanf	libc.a
*	_strcat	libc.a
*	_strchr	libc.a
*	_strcmp	libc.a
*	_strcoll	libc.a
*	_strcpy	libc.a
*	_strcspn	libc.a
*	_strerror	libc.a
*	_strlen	libc.a
*	_strncat	libc.a
*	_strncmp	libc.a

Index	Definition name	Library
*	_strncpy	libc.a
*	_strpbrk	libc.a
*	_strrchr	libc.a
*	_strspn	libc.a
*	_strstr	libc.a
*	_strtod	libc.a
*	_strtok	libc.a
*	_strtol	libc.a
*	_strtoul	libc.a
*	_strxfrm	libc.a
*	_tan	libm.a
*	_tanh	libm.a
*	_tolower	libc.a
*	_toupper	libc.a
*	_ungetc	libc.a
*	_unlink	libc.a
*	_vfprintf	libc.a
*	_vprintf	libc.a
*	_vsprintf	libc.a
*	_wcstombs	libc.a

Index	Definition name	Library
*	_wctomb	libc.a
*	_write	libc.a

Support Library and Math Library Descriptions

The remainder of this chapter describes the support and math library functions. Functions declared in the **math.h** include file are found in the math library archive files **libm.a**, **libmpi.a**, **libmpc.a**, **libm881.a**, **libm881pi.a**, and **libm881pc.a**. All other functions are found in the support library archive files **libc.a**, **libcpi.a**, and **libcpc.a**.

Note

The **open**, **close**, **read**, **write**, **lseek**, **unlink**, **exit**, **_exit**, **_getmem**, and **sbrk** functions have execution environment dependencies; therefore, these libraries are described in the "Environment-Dependent Routines" chapter.

abs, labs

Return Integer Absolute Value

Synopsis

```
#include <stdlib.h>

int abs (int i);

long int labs (long int i);
```

Description

Abs returns the absolute value of its integer operand.

Labs is similar to *abs* except that the argument and the returned value each have type **long int**.

Warnings

In two’s-complement representation, the absolute value of the negative integer with the largest magnitude is undefined. This error is ignored.

See Also

floor.

assert

Put Diagnostics into Programs

Synopsis

#include <assert.h>

void assert (const char *expression);

Description

The *assert* macro puts diagnostics into programs. When it is executed, if *expression* is false (equal to zero), the *assert* macro writes information about the particular call that failed (including the text of the argument, the name of the source file, and the source line number – the latter are respectively the values of the preprocessing macros **__FILE__** and **__LINE__**) on the standard error file in the format shown below. It then calls the *_exit* function.

```
Assertion failed:  <expression>,  file  <__FILE__>,  line  <__LINE__>
```

Diagnostics

When the assert.h header file is included and the macro **NDEBUG** is defined, the *assert* macro will be defined to do nothing. This allows you to compile your code with or without the *assert* checking by simply defining or undefining the macro **NDEBUG**. *Assert* returns no value.

See Also

_exit.

atexit

Call Function at Program Termination

Synopsis

#include <stdlib.h>

int atexit (void (*func)(void));

Description

Atexit will register the *func* function to be called without arguments at normal program termination. Up to 32 separate function registrations can be performed.

Diagnostics

Atexit returns zero if the registration succeeds, or non-zero if it fails.

See Also

exit.

bsearch

Binary Search a Sorted Table

Synopsis

```
#include <stdlib.h>

void *bsearch (
const void *key,
const void *base,
size_t nel, size_t size,
int (*compar)(const void *, const void *));
```

Description

Bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *Key* points to a datum instance to be sought in the table. *Base* points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero as accordingly the first argument is to be considered less than, equal to, or greater than the second.

Notes

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type void pointer. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared. Although declared as void pointer type, the value returned should be cast into type pointer-to-element.

Example

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

This code fragment reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

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

#define        TABSIZE        1000

struct node {                      /* these are stored in the table */
        char *string;
        int length;
};
struct node table[TABSIZE];        /* table to be searched */
        .
        .
        .
{
        struct node *node_ptr, node;
        int node_compare( );  /* routine to compare 2 nodes */
        char str_space[20];   /* space to read string into */
        .
        .
        .
        node.string = str_space;
        while (scanf("%s", node.string) != EOF) {
                node_ptr = (struct node *)bsearch((void *)(&node),
                           (void *)table, TABSIZE,
                           sizeof(struct node), node_compare);
                if (node_ptr != NULL) {
                        (void)printf("string = %20s, length = %d\n",
                                node_ptr->string, node_ptr->length);
                } else {
                        (void)printf("not found: %s\n", node.string);
                }
        }
}
/*      This routine compares two nodes based on an
        alphabetical ordering of the string field.     */
int
node_compare(node1, node2)
struct node *node1, *node2;
{
        return strcmp(node1->string, node2->string);
}
```

See Also

qsort.

Diagnostics

A NULL pointer is returned if the key cannot be found in the table.

Bugs

A random entry is returned if more than one entry matches the selection criteria.

div, ldiv

Divide Functions

Synopsis

#include <stdlib.h>

div_t div (int numer, int denom);

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

Description

Div computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the sign of the quotient is that of the mathematical quotient, and the magnitude of the quotient is the largest integer less than the magnitude of the mathematical quotient. If the result cannot be represented, the behavior is undefined.

Ldiv is similar to *div* except that the arguments and members of the returned structure (which has type **ldiv_t**) all have type **long int**.

Diagnostics

The *div* function returns a structure of type **div_t**, comprising both the quotient and the remainder. The structure is defined by **stdlib.h** as shown below.

```
typedef struct {
                int  quot;      /* Quotient */
                int  rem;       /* Remainder */
        } div_t;

typedef struct {
                long int  quot; /* Quotient */
                long int  rem;  /* Remainder */
        } ldiv_t;
```

exp

Exponential Functions

Synopsis

#include <math.h>

double exp (double x);

Description

Exp returns $e^{x.}$

Diagnostics

Exp sets *errno* to **ERANGE** and returns **HUGE_VAL** when the correct value would overflow, or 0 when the correct value would underflow. In addition to *errno*, bits in a global status flag or in the floating point unit floating-point status register are set when error conditions arise.

The error-handling is done by the run-time **_fp_error** routine.

See Also

_fp_error, **_get_fp_status**, "Behavior of Math Library Functions" chapter.

fclose, fflush

Close or Flush a Stream

Synopsis

#include <stdio.h>

int fclose (FILE *stream);

int fflush (FILE *stream);

Description

Fclose causes any buffered data for the named *stream* to be written out, and the *stream* to be closed. Buffers allocated by the standard input/output system are freed.

Fclose is performed automatically for all open files upon calling **exit**.

Fflush causes any buffered data for the named *stream* to be written to that file. If the argument is NULL, then all open files are flushed. The *stream* or *streams* remain open.

Diagnostics

These functions return 0 for success, and **EOF** if any error (such as trying to write to a file that has not been opened for writing) was detected.

See Also

close, **exit**, **fopen**, **setbuf**.

ferror, feof, clearerr

Stream Status Inquiries

Synopsis

#include <stdio.h>

int ferror (FILE *stream);

int feof (FILE *stream);

void clearerr (FILE *stream);

Description

Ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*, otherwise zero. Unless cleared by *clearerr*, or unless the specific *stdio* routine so indicates, the error indication lasts until the stream is closed.

Feof returns non-zero when **EOF** has previously been detected reading the named input *stream*, otherwise zero.

Clearerr resets the error indicator and **EOF** indicator to zero on the named *stream*.

Note

These functions are implemented as macros and functions. To use a function instead of a macro, **#undef** the macro before function invocation.

See Also

open, **fopen**.

fgetpos, fseek, fsetpos, rewind, ftell

Position File Pointer

Synopsis

#include <stdio.h>

int fgetpos (FILE *stream, fpos_t *pos);

int fseek (FILE *stream, long offset, int ptrname);

int fsetpos (FILE *stream, const fpos_t *pos);

long ftell (FILE *stream);

void rewind (FILE *stream);

Description

Fgetpos stores the current value of the file pointer on the *stream* in the object pointed to by *pos*. The value stored contains unspecified information usable by the *fsetpos* function for repositioning the stream to its position at the time of the call to the *fgetpos* function.

Fsetpos sets the file pointer for the *stream* to the value of the object pointed to by *pos* which is a value returned by an earlier call to *fgetpos* on the same stream.

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according as *ptrname* has the value **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**.

Rewind (stream) is equivalent to *(void) fseek* (stream , 0L, SEEK_SET).

Fsetpos, *fseek*, and *rewind* clear the end-of-file indicator and undo any effects of the *ungetc* function on the same stream. After an *fsetpos*, *fseek*, or *rewind* call, the next operation on an update stream may be either input or output. *Rewind* also does an implicit **clearerr** call.

Ftell returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

See Also

lseek, **fopen**, **ungetc**.

Diagnostics

The *fgetpos* and *fsetpos* functions return zero if successful; otherwise, they return non-zero and *errno* is set to a non-zero value.

Fseek returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen*; in particular, *fseek* may not be used on a terminal.

Ftell returns –1 for error conditions and sets *errno* to a non-zero value. If either the argument to *ftell* is NULL or if the file is not open, then *ftell* sets **errno** to EBADF.

Warning

In UNIX-base operating sytems, the offset returned by *ftell* is measured in bytes, and a program may seek to positions relative to that offset. Portability to non-UNIX systems requires that an offset be used by *fseek* directly. Do not use the offset in calculations—the offset might not be measured in bytes.

floor, ceil, fmod, frem, fabs

Floor, Ceiling, Remainder, and Absolute Value

Synopsis

#include <math.h>

double floor (double x);
double ceil (double x);
double fmod (double x, double y);
double frem (double x, double y);
double fabs (double x);

Description

Floor returns the largest integer (as a double-precision number) not greater than *x*.

Ceil returns the smallest integer (as a double-precision number) not less than *x*.

Fmod returns the floating-point remainder of the division of *x* by *y*: NaN if *y* is zero or **+/-HUGE_VAL** if *x/y* would overflow; otherwise the number *f* with the same sign as *x*, such that $x = iy + f$ for some integer *i*, and $|f| < |y|$.

Frem is the same as *fmod* except that the remainder is computed in round-to-nearest mode, and the result may have a different sign than *x*. For example:

```
fmod (x, y) = x – (y*i)     Where i = (int) (x/y)

frem (x, y) = x – (y*i)     Where i = (int) (x/y + 0.5)

fmod (5.2, 10) = 5.2 – (10*0) = 5.2
frem (5.2, 10) = 5.2 – (10*1) = –4.8
```

Fabs returns the absolute value of *x*, |*x*|; *errno* is set whenever an exception condition occurs.

See Also

abs, "Behavior of Math Library Functions" chapter.

fopen, freopen

Open or Re-Open a Stream File

Synopsis

```
#include <stdio.h>

FILE *fopen (
const char *file_name,
const char *type);

FILE *freopen (
const char *file_name,
const char *type,
FILE *stream);
```

Description

Fopen opens the file named by *file_name* and associates a *stream* with it. *Fopen* returns a pointer to the FILE structure associated with the *stream*.

File_name points to a character string that contains the name of the file to be opened.

Type is a character string having one of the following values:

"r", "rb"	Open for reading.
"w", "wb"	Truncate or create for writing.
"a", "ab"	Append; open for writing at end of file, or create for writing.
"r+", "rb+", "r+b"	Open for update (reading and writing).
"w+", "wb+", "w+b"	Truncate or create for update.
"a+", "ab+", "a+b"	Append; open or create for update at end-of-file.

A character "b" in the type string signifies that the file is a binary file. In this implementation, the presence or absence of the "b" has no effect.

Freopen substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. *Freopen* returns a pointer to the FILE structure associated with *stream*.

Freopen is typically used to attach the preopened *streams* associated with **stdin**, **stdout**, and **stderr** to other files.

When a file is opened for update (i.e., the character "+" is present in the *type* string), both input and output may be done on the resulting *stream*. However, input may not be directly followed by output unless there is an intervening call to *fflush* or to one of the file positioning functions (*rewind, fseek, fsetpos*). The same is true for following output directly with input.

When a file is opened for append (i.e., the character "a" is present in the *type* string), information already present in the file cannot be overwritten. *Fseek* may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. Undefined behavior will occur if the file is also opened for update and the preceding rules for update mode are not followed.

See Also

open, **fclose**, **fseek**.

Diagnostics

Fopen and *freopen* return a NULL pointer if *file-name* cannot be accessed, if there are too many open files, or if the arguments are incorrect.

_fp_error

Floating-Point Error Functions

Synopsis

#include <fp_control.h>

void _clear_fp_status (void);

int _get_fp_status (void);

void _set_fp_control (int mode);

int _get_fp_control (void);

Description

Technically, *_fp_error* is a run-time routine in that it is only called from other run-time library and math library functions. Its purpose is to simulate the exception processing that is present on the FPU. Therefore, *_fp_error* is referenced only when the libraries **lib.a** and **libm.a** are loaded.

When called, *_fp_error* inspects a global control flag to see if the trap bit associated with the current exception is set. If the bit is set, *_fp_error* calls the **trap()** routine which composes an error message and traps into the monitor program to display the message. If the bit is not set, *_fp_error* updates a global status flag to reflect the exception type and returns a value defined by the IEEE Floating Point Standard 754 (see the "Behavior of Math Library Functions" chapter).

FP functions (68020, 68030, 68040)

_clear_fp_status either clears the global status flag or the floating-point status register of the FPU.

_get_fp_status returns either the global status flag or the floating-point status register of the FPU.

_set_fp_control sets either the global control flag or the floating-point control register of the FPU to *mode*.

_get_fp_control returns either the global control flag or the floating-point control register of the FPU.

Default mode

The 68000 and 68332 libraries always perform operations in double precision and round to nearest. By default, trapping is enabled on all floating-point exceptions except inexact results.

For the 68020 and 68030, the default mode of operation within the FPU libraries is to perform operations in extended precision, round to nearest, and to trap on all floating-point exceptions except inexact results. The libraries always perform operations in double precision and round to nearest. By default, trapping is enabled on all floating-point exceptions except inexact results.

For the 68040, the default mode of operation is to perform operations in extended precision, round to nearest, and to trap on all floating-point exceptions except inexact results.

Note Because of the various rounding and precision modes available with the 68881/2 FPU, the 68881/2 libraries may yield different results than the other libraries.

Mode macros

For the 68020 and 68030, the following macros may be OR'ed together to form *mode* when invoking *_set_fp_control*. All except NOTRAP are only meaningful in 68881/2 libraries.

RNDNEAR	Round to nearest.
RNDZERO	Round to zero, or truncate.
RNDNEGINF	Round towards minus infinity.
RNDPOSINF	Round towards plus infinity.
SGLPREC	Perform operations in single precision.
DBLPREC	Perform operations in double precision.
EXTPREC	Perform operations in extended precision.
NOTRAP	Disable all traps.

BSUN	Trap on IEEE non-aware branches.
INEXACTD	Trap on inexact decimal input.

For the 68040, the following macros may be OR'ed together to form *mode* when invoking *_set_fp_control*:

RNDNEAR	Round to nearest.
RNDZERO	Round to zero, or truncate.
RNDNEGINF	Round towards minus infinity.
RNDPOSINF	Round towards plus infinity.
SGLPREC	Perform operations in single precision.
DBLPREC	Perform operations in double precision.
EXTPREC	Perform operations in extended precision.
NOTRAP	Disable all traps.
BSUN	Trap on IEEE non-aware branches.
INEXACTD	Trap on inexact decimal input.

FP functions (68000 and 68332)

For the 68000 and 68332,

_clear_fp_status clears the global status flag.

_get_fp_status returns the global status flag.

_set_fp_control sets the global control flag to *mode*.

_get_fp_control returns the global control flag.

Mode macros (68000 and 68332)

For the 68000 and 68332, the following macros may be OR’ed together to form *mode* when invoking *_set_fp_control*:

NOTRAP	Disable all traps.
INEXACT	Trap on inexact result.
DIVZERO	Trap on division by zero.
UNDERFLOW	Trap on underflow.
OVERFLOW	Trap on overflow.
OPERROR	Trap on operand error.
SIGNAN	Trap on detection of signaling NaN.
PLOSS	Trap on loss of precision (applies when the FPU is not used).

Macros for interpreting status

The following macros may be used when inspecting the return value from *_get_fp_status*:

NOERRORS	No errors have been detected since the last invocation of *_clear_fp_status*.

INEXACTD (68020/30/40)
BSUN (68020/30/40)
INEXACT
DIVZERO
UNDERFLOW
OVERFLOW
OPERROR
SIGNAN
PLOSS

Example

You may change the control word without respecifying all of the different categories. This can be done by using the current value of the control variable and using masking. For example, the following function call turns on divide-by-zero trapping without altering any of the other control flags:

```
_set_fp_control(_get_fp_control() | DIVZERO );
```

The next example turns off the overflow and underflow traps:

```
_set_fp_control(_get_fp_control() &
        ~(UNDERFLOW  | OVERFLOW ));
```

fread, fwrite

Buffered Binary I/O to Stream

Synopsis

```
#include <stdio.h>

size_t fread (void *ptr, size_t size,
            size_t nitems, FILE *stream);

size_t fwrite (const void *ptr, size_t size,
             size_t nitems, FILE *stream);
```

Description

Fread copies, into an array pointed to by *ptr*, *nitems* items of data from the named input stream, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. *Fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. *Fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *Fread* does not change the contents of *stream*.

Fwrite appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream*. *Fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. *Fwrite* does not change the contents of the array pointed to by *ptr*.

The argument *size* is typically *sizeof(*ptr)* where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than **void** it should be cast into a pointer to **void**.

See Also

read, **write**, **fopen**, **getc**, **gets**, **printf**, **putc**, **puts**, **scanf**.

Diagnostics

Fread and *fwrite* return the number of items read or written. If *size* or *nitems* is zero, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

frexp, ldexp, modf

Return Mantissa and Exponent

Synopsis

```
#include <math.h>

double frexp (double value, int *eptr);

double ldexp (double value, int *exp);

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

Description

Every non-zero number can be written uniquely as *x* * 2*n* where the "mantissa" (fraction) *x* is in the range 0.5 <= |*x*| < 1.0, and the "exponent" *n* is an integer.

Frexp returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*. If *value* is zero, both results returned by *frexp* are zero.

Ldexp returns the quantity *value* * 2exp.

Modf returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

Diagnostics

If *ldexp* would cause overflow, **+/-HUGE_VAL** is returned (according to the sign of *value*), and *errno* is set to **ERANGE**. If *ldexp* would cause underflow, zero is returned and *errno* is set to **ERANGE**.

See Also

_fp_error, "Behavior of Math Library Functions" chapter.

getc, getchar, fgetc

Get Character from Stream

Synopsis

```
#include <stdio.h>

int getc (FILE *stream);
int getchar (void);
int fgetc (FILE *stream);
```

Description

Getc returns the next character (i.e., byte) from the named input *stream*, as an integer. It also moves the file pointer, if defined, ahead one character in *stream*. *Getchar* is defined as *getc(stdin)*. *Getc* is a macro and so cannot be used if a function is necessary; for example one cannot have a function pointer point to it. *Getchar* is implemented as a macro and as a function. To use a function instead of a macro, **#undef** the macro before function invocation.

Fgetc behaves like *getc*, but is a function rather than a macro. *Fgetc* runs more slowly than *getc*, but it takes less space per invocation and its name can be passed as an argument to a function.

See Also

fclose, **ferror**, **fopen**, **fread**, **gets**, **putc**, **scanf**.

Diagnostics

These functions return the constant **EOF** at end-of-file or upon an error.

Warning

If the integer value returned by *getc*, *getchar*, or *fgetc* is stored into a character variable and then compared against the integer constant **EOF**, the comparison may never succeed, because sign-extension of a character on widening to integer is machine-dependent.

Bugs

Because it is implemented as a macro, *getc* treats incorrectly a *stream* argument with side effects. In particular, **getc(*f++)** does not work sensibly. *Fgetc* should be used instead.

gets, fgets

Get a String from a Stream

Synopsis

#include <stdio.h>

char *gets (char *s);

char *fgets (char *s, int n, FILE *stream);

Description

Gets reads characters from the standard input stream, *stdin,* into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

Fgets reads characters from the *stream* into the array pointed to by *s*, until *n*-1 characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

See Also

ferror, **fopen**, **fread**, **getc**, **puts**, **scanf**.

Diagnostics

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned, and the contents of *s* are indeterminate. Otherwise *s* is returned.

interpolateS, interpolateSN, interpolateU, interpolateUN

68332 Data Register Interpolate

Synopsis

```
#include <m68332.h>

int interpolateS (int y1, int y2, char x);
int interpolateSN (int y1, int y2, char x);
unsigned int interpolateU (unsigned int y1, unsigned int y2, char x);
unsigned int interpolateUN (unsigned int y1, unsigned int y2, char x);
```

Description

These functions interpolate between two values as if they were two table entries. These functions may be used with the *table* routines to model multidimensional functions. The functions are expanded to the 68000 table lookup and interpolate instruction, using the data register interpolate mode, as follows:

interpolateS is expanded to TBLS.L (signed, rounded).
interpolateSN is expanded to TBLSN.L (signed, unrounded).
interpolateU is expanded to TBLU.L (unsigned, rounded).
interpolateUN is expanded to TBLUN.L (unsigned, unrounded).

Byte *x* is used as an interpolation fraction. Rounded results are determined as follows:

$$y1 + ((y2 - y1) * x) / 256$$

and unrounded results are determined by:

$$y1*256 + ((y2 - y1) * x)$$

The file **m68332.h** must be included for the table functions to be expanded in-line.

See Also

table, also the assembly language manual.

isalpha, isupper, islower, ...

Classify Characters

Synopsis

#include <ctype.h>

int isalpha (int c);

. . .

Description

These routines classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. These routines are implemented both as macros and functions. To use a function instead of a macro, **#undef** the macro before function invocation.

isalpha — *c* is a letter.

isupper — *c* is an upper-case letter.

islower — *c* is a lower-case letter.

isdigit — *c* is a digit [0-9].

isxdigit — *c* is a hexadecimal digit [0-9], [A-F] or [a-f].

isalnum — *c* is an alphanumeric (letter or digit).

isspace — *c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.

ispunct — *c* is a printing character that is neither a control character nor an alphanumeric character nor a space.

isprint — *c* is a printing character, code 040 (space) through 0176 (tilde).

isgraph — *c* is a printing character, like *isprint* except false for space.

iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

Diagnostics

If the argument to any of these macros is not in the domain of the function, the result is undefined. The domain for these functions is the integer values [0, 255] and **EOF**.

localeconv

Locale Conversion

Synopsis

#include <locale.h>

struct lconv *localeconv (void);

Description

localeconv sets the components of an object of type *struct lconv* to the appropriate numeric quantity formatting values for the current locale.

Within the structure *lconv*, members of type **char *** point to strings. Any char pointer, except **char *decimal_point** may point to a null string ("") to indicate that the value is either not available in the current locale or of zero length in the current locale.

The following are members of the **lconv** structure:

char *decimal_point

is the decimal point character used to format non-monetary quantities.

char *thousands_sep

is used to separate groups of digits before the decimal point in non-monetary quantities.

char *grouping

is a string, the elements of which indicate the size of each group of digits in formatted non-monetary quantities.

char *int_curr_symbol

is the international currency symbol used in the current locale. The first three characters in this string contain the alphabetic international currency symbol in accordance with *ISO 4217 Codes for the Representation of Currency and Funds*. The fourth character is (last before the null terminator) is the character used to separate the currency symbol from the monetary quantity.

char *currency_symbol

is the local currency symbol for the current locale.

char *mon_decimal_point

is the decimal point used to format the monetary values.

char *mon_thousands_sep

is the separator for groups of digits before the decimal point in the monetary values.

char *mon_grouping

is a string, the elements of which indicate the size of each group of digits in formatted monetary quantities.

char *positive_sign

is the string used to signify non-negative formatted monetary values.

char *positive_sign

is the string used to signify negative formatted monetary values.

char int_frac_digits

is the number of fractional digits (after the decimal point) to display in an internationally formatted monetary value.

char frac_digits

is the number of fractional digits (after the decimal point) to display in a formatted monetary value.

char p_cs_precedes

for a formatted non-negative monetary value, is set to one if the **currency_symbol** precedes the value or set to zero if the currency_symbol follows the value.

char p_sep_by_space

for a formatted non-negative monetary value, is set to one if the **currency_symbol** is separated from the value by a space and set to zero if it is not separated from the value by a space.

char n_cs_precedes

for a formatted negative monetary value, is set to one if the **currency_symbol** precedes the value or set to zero if the currency_symbol follows the value.

char p_sep_by_space

for a formatted negative monetary value, is set to one if the **currency_symbol** is separated from the value by a space and set to zero if it is not separated from the value by a space.

char p_sign_posn

is a value indicating the positioning of the negative sign for a formatted non-negative monetary value.

char n_sign_posn

is a value indicating the positioning of the negative sign for a formatted negative monetary value.

The elements *grouping* and *mon_grouping* specify the grouping of digits in non-monetary and monetary quantities. Both strings are strings of grouping counts. The first element of the string, say s[0], unless it is CHAR_MAX, is the number of digits to group before the first separator character. s[1], unless it is zero or CHAR_MAX, is the number of digits to group after grouping s[0] digits. s[2], unless it is zero or CHAR_MAX, is the number of digits to group after s[0] digits and s[1] digits have been grouped. And so on. If s[i] is zero, then the value in s[i-1] is the grouping value for all subsequent digits. If s[i] is CHAR_MAX, then no further grouping is performed.

The value of either *p_sign_posn* and *n_sign_posn* is interpreted in the following way:

0	Parentheses surround the quantity and *currency_symbol*.
1	The sign string precedes the quantity and *currency_symbol.*
2	The sign string follows the quantity and *currency_symbol.*
3	The sign string immediately precedes the *currency_symbol*.
4	The sign string immediately follows the *currency_symbol*.

Diagnostics

The *localeconv* routine returns a pointer to the filled object. The returned structure is not to be modified directly by the program, but may be overwritten by further calls to localeconv. In addition, calls to *setlocale* with the categories LC_ALL, LC_MONETARY, and LC_NUMERIC may overwrite the contents of the structure.

Note

The locale supported by the libraries is the "C" locale. *localeconv* will return the "C" locale only. The following table lists the return values for the various structure elements.

Additionally, there is a macro **MB_CUR_MAX** defined in **stdlib.h** that returns the maximum number of bytes a multi-byte character could have in the current locale. Because multi-byte characters are not supported, this macro always returns one.

See Also

setlocale

Table 7-2. Element Values Returned by ***localeconv***

Element	**Returned Value**
char *decimal_point	"."
char *thousands_sep	""
char *grouping	""
char *int_curr_symbol	""
char *currency_symbol	""
char *mon_decimal_point	""
char *mon_thousands_sep	""
char *mon_grouping	""
char *positive_sign	""
char *negative_sign	""
char int_frac_digits	""
char frac_digits	CHAR_MAX
char p_cs_precedes	CHAR_MAX
char p_sep_by_space	CHAR_MAX
char n_cs_precedes	CHAR_MAX
char n_sep_by_space	CHAR_MAX
char p_sign_posn	CHAR_MAX
char n_sign_posn	CHAR_MAX

log, log10

Logarithm Functions

Synopsis

#include <math.h>

double log (double x);

double log10 (double x);

Description

Log returns the natural logarithm of *x*. The value of *x* must be positive.

Log10 returns the logarithm base ten of *x*. The value of *x* must be positive.

Diagnostics

Log and *log10* return **-HUGE_VAL** and set *errno* to **EDOM** when *x* is negative. *Log* and *log10* return an NaN and set *errno* to ERANGE when *x* is zero. The error action is determined by the bits of the global control flag or the FPU floating-point control register.

These error-handling procedures may be changed with the function **_fp_error**.

See Also

_fp_error, "Behavior of Math Library Functions" chapter.

malloc, free, realloc, calloc

Main Memory Allocator

Synopsis

#include <stdlib.h>

void *malloc (size_t size);

void free (void *ptr);

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

void *calloc (size_t nelem, size_t elsize);

Description

Malloc and *free* provide a simple general-purpose memory allocation package. *Malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation.

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free.*

Malloc calls *_getmem* to get more memory when there is no suitable space already free.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If the size argument to realloc is zero, then a free operation is done.

If no free block of *size* bytes is available in the storage arena, then *realloc* will ask *malloc* to enlarge the arena by *size* bytes and will then move the data to the new space.

Calloc allocates space for an array of *nelem* elements of size *elsize.* The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

See Also

_getmem. (Described in the "Environment-Dependent Routines" chapter.)

Diagnostics

Malloc, *realloc* and *calloc* return a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When this happens the block pointed to by *ptr* may be destroyed.

mblen, mbstowcs, mbtowc, wcstombs, wctomb, strxfrm

Multi-byte Character Operations

Synopsis

#include <stdlib.h>

int mblen (const char *s, size_t n);

size_t mbstowcs (wchar_t *pwcs, const char *s, size_t n);

int mbtowc (wchar_t *pwc, const char *s, size_t n);

size_t wcstombs (char *s, const wchar_t *pwcs, size_t n);

int wctomb (char *s, wchar_t wchar);

#include <string.h>

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

Description

mblen, because multi-byte characters are not supported, returns zero if the first argument is NULL—without regard to the value of n. If the first argument is not NULL, then *mblen* returns negative one if *n* is zero or returns one if *n* is nonzero.

mbstowcs copies n multi-byte characters from the second argument into the first, transforming each multi-byte character into its wide character representation. Because multi-byte characters are not supported, *mbtowcs* copies *n* bytes from the second argument to the first while transforming each byte to its wide character representation. Transformation is accomplished by moving the character value into the least significant byte and zero-filling the remaining bytes of the wide character. If there is room left in the first argument after copying all bytes, *mbstowcs* appends a null terminating character to the first argument. *mbstowcs* returns the number of multi-bytes copied (which, in this implementation, is the number of bytes copied). That number may be less than *n* if a null character is found in the second argument before *n* bytes are read.

mbtowc transforms the multi-byte character from the second argument into its wide character representation and places it into the first argument. *mbtowc* uses at most *n* bytes from the second argument. Because multi-byte characters are not supported, *mbtowc* copies *n* characters from the second argument into the first and transforms each character as it is copied by moving the character value into the least significant byte and zero-filling the remaining bytes of the wide character. *mbtowc* returns zero if the second argument is NULL or returns one if the second argument is not NULL.

wcstombs copies *n* wide characters from the second argument into the first while transforming each wide character into its multi-byte character representation. Because multi-byte characters are not supported, *wctombs* copies at most *n* characters from the second argument into the first while transforming each character by copying just the least significant byte of the wide character. If there is room in the first argument after copying, *wcstombs* appends a null terminator. *wcstombs* returns the number of bytes copied, which may be less than *n* if a null terminating character is found in the second string before *n* bytes are read.

wctomb transforms the wide character pointed to by the second argument into a multi-byte character and places it in the first argument. The wide character will be represented by at most MB_CUR_MAX characters in the multi-byte character. Because multi-byte characters are not supported, MB_CUR_MAX is always one and therefore the wide character transformed into a single character. The transformation is accomplished by copying the least significant byte of the wide character into the char. *wctomb* returns zero if the second argument is NULL or returns one if the second argument is not NULL.

strxfrm, because multi-byte characters are not supported, simply does a byte-by-byte copy from s2 to s1 of up to *n* characters.

Note

In addition to the multi-byte character operations, the macro **MB_CUR_MAX** returns the maximum number of bytes a multi-byte character could have in the current locale. Because multi-byte characters are not supported, this macro always returns one.

memchr, memcmp, memcpy, memmove, memset

Memory Operations

Synopsis

#include <string.h>

void *memchr (const void *s, int c, size_t n);
int memcmp (const void *s1, const void *s2, size_t n);
void *memcpy (void *s1, const void *s2, size_t n);
void *memmove (void *s1, const void *s2, size_t n);
void *memset (void *s, int c, size_t n);

Description

These functions operate efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Memchr returns a pointer to the first occurrence of character **c** in the first **n** characters of memory area **s**, or a NULL pointer if **c** does not occur.

Memcmp compares its arguments, looking at the first **n** characters only, and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**. (*n* equal to zero yields equality.) In some operating systems, *memcmp* uses **unsigned char** for character comparisons. This may not be true for other implementations.

Memcpy copies **n** characters from memory area **s2** to **s1**. It returns **s1**.

Memmove works like *memcpy* except that *memmove* handles overlapping moves properly.

Memset sets the first **n** characters in memory area **s** to the value of character **c**. It returns **s**.

Bugs

Strcpy and *memcpy* may fail for overlapping moves; use *memmove* instead.

See Also

strchr, **strrchr**, **strcmp**, **strncmp**, **strcpy**, **strncpy**.

perror, errno

System Error Messages

Synopsis

#include <stdio.h>

void perror (const char *s);

#include <errno.h>

extern int errno;

Description

Perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

The value of *errno* might not be what you expect if your program uses multitasking; *errno* can be overwritten by some library routines.

See Also

strerror.

pow

Power Function

Synopsis

#include <math.h>

double pow (double x, double y);

Description

Pow returns x^y. If *x* is zero, *y* must be positive. If *x* is negative, *y* must be an integer.

Diagnostics

Pow returns NaN (Not a Number) and sets *errno* to **EDOM** when *x* is 0 and *y* is non-positive, or when *x* is negative and *y* is not an integer. The error action is determined by the bits of the global control flag or the FPU floating-point control register. When the correct value for *pow* would overflow or underflow, *pow* returns **+/-HUGE_VAL** or 0 respectively, and sets *errno* to **ERANGE**.

These error-handling procedures may be changed with the function **_fp_error**.

See Also

_fp_error, "Behavior of Math Library Functions" chapter.

printf, fprintf, sprintf

Print Formatted Output

Synopsis

#include <stdio.h>

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

int fprintf (FILE *stream, const char *format, ...);

int sprintf (char *s, const char *format, ...);

Description

Printf places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places "output", followed by the null character (**0**), in consecutive bytes starting at *s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the **0** in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *arg*s under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more *arg*s. The results are undefined if there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, the excess *arg*s are evaluated but ignored.

The behavior of the sprintf function is undefined if the destination array is also one of the other arguments. This undefined behavior of sprintf is particularly important because the behavior has changed between versions of the HP cross compilers.

Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

> Zero or more *flags*, which modify the meaning of the conversion specification.
>
> An optional decimal digit string specifying a minimum *"field width"*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '-', described below, has been given) to the field width. If the field width for a conversion is preceded by a 0, the padding is done with zeros instead of spaces.

A *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (**.**) followed by a decimal digit string; a null digit string is treated as zero.

An optional **l** (ell) specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*, or an optional **h** specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a short integer *arg*. A "%ln" format means that the argument is a pointer to a long integer and a "%hn" format means that the argument is a pointer to a short integer.

An optional **L** specifies that a following **e**, **E**, **f**, **g**, or **G** conversion character applies to a long double *arg*.

An **l** or **L** before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*****) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *arg*s specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

–	The result of the conversion will be left-justified within the field.
+	The result of a signed conversion will always begin with a sign (**+** or **-**).
blank	If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and **+** flags both appear, the blank flag will be ignored.
#	This flag specifies that the value is to be converted to an "alternate form." For **c**, **d**, **i**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** or **X** conversion, a non-zero result will have **0x** or **0X** prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,i,o,u,x,X	The integer *arg* is converted to signed decimal (**d** or **i**), unsigned octal, unsigned decimal, or hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prefixing a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

f

The double *arg* is converted to decimal notation in the style "[**-**]ddd**.**ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no decimal point appears.

e, E

The double *arg* is converted in the style "[-]d**.**ddd**e+/-**ddd", where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.

g, G

The double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than –4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

c

The character *arg* is printed.

s

The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (**0**) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.

p

The *arg* is taken to be a pointer to **void**. The value of the pointer is converted to a sequence of printable characters, in the same manner as **%x**.

n

The *arg* is taken to be a pointer to an integer into which is written the number of characters written to the output stream so far by this call to *printf*. No argument is converted.

`%` Print a **`%`**; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if **putc** had been called.

Examples

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %d:%.2d", weekday, month, day, hour,
min);
```

To print *pi* to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

The value of **`string1`** is undefined after the following line of code:

```
sprintf (string1, "%s %d", string1, integer1);
```

See Also

putc, **scanf**, **vprintf**.

putc, putchar, fputc

Put a Character on a Stream

Synopsis

#include <stdio.h>

int putc (int c, FILE *stream);

int putchar (int c);

int fputc (int c, FILE *stream);

Description

Putc writes the character *c* onto the output *stream* (at the position where the file pointer, if defined, is pointing). *Putchar*(c) is defined as *putc* (c, stdout). *Putc* is implemented as a macro; *putchar* is implemented as a macro and as a function. To use a function instead of a macro, **#undef** the macro before function invocation.

Fputc behaves like *putc*, but is a genuine function rather than a macro; it may therefore be used as an argument. *Fputc* runs more slowly than *putc*, but it takes less space per invocation and its name can be passed as an argument to a function.

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file. The standard error output stream *stderr* is by default unbuffered, but use of *freopen* (see **fopen**) will cause it to become buffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing as soon as the line is completed (that is, as soon as a new-line character is written or input is requested). *Fflush* can also be used to explicitly write the buffer. **Setbuf** or **setvbuf** may be used to change the stream's buffering strategy.

These routines do not have the means to determine if a file is associated with a terminal. Therefore, files are fully buffered, except for *stdin* and *stdout* which are set to line-buffered by the **_startup** routine and *stderr* which is not buffered.

See Also

fclose, **ferror**, **fopen**, **fwrite**, **getc**, **fread**, **printf**, **puts**, **setbuf**.

Diagnostics

On success, these functions each return the value they have written. On failure, they return the constant **EOF**. This will occur if the file *stream* is not open for writing or if the output file cannot be increased.

Line buffering may cause confusion or malfunctioning of programs which use standard I/O routines but use **read** themselves to read from standard input. In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to **fflush** the standard output before going off and computing so that the output will appear.

Bugs

Because it is implemented as a macro, *putc* treats incorrectly a *stream* argument with side effects. In particular, **putc(c, *f++);** doesn't work sensibly. *Fputc* should be used instead.

puts, fputs

Put a String on a Stream

Synopsis

#include <stdio.h>

int puts (const char *s);

int fputs (const char *s, FILE *stream);

Description

Puts writes the null-terminated string pointed to by *s*, followed by a new-line character, to the standard output stream *stdout*.

Fputs writes the null-terminated string pointed to by *s* to the named output *stream*.

Neither function writes the terminating null character. Note that *puts* appends a new-line character, but *fputs* does not.

Diagnostics

If the routine is successful, *puts* and *fputs* both return the number of characters written. In the case of *puts*, the return value includes the implied newline character which means that the return value will equal the length of the argument string + 1.

See Also

ferror, **fopen**, **fread**, **printf**, **putc**.

qsort

Table Sorting Routine

Synopsis

```
#include <stdlib.h>

void qsort (
void *base,
size_t nel, size_t size,
int (*compar)(const void *, const void *));
```

Description

Base points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function passed as *compar* must return an integer less than, equal to, or greater than zero as a consequence of whether its first argument is to be considered less than, equal to, or greater than the second. This is the same return convention that *strcmp* uses.

Notes

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared. The order in the output of two items which compare as equal is unpredictable.

See Also

bsearch.

rand, srand

Simple Random Number Generator

Synopsis

```
#include <stdlib.h>

int rand (void);
```

void srand (unsigned int seed);

Description

Rand uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to 2^{15}-1.

Srand can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

Note

The spectral properties of *rand* leave a great deal to be desired. These functions use a global variable to seed the random number generator. Calling one of these routines from an interrupt routine will cause the random number sequence to be non-repeatable.

remove

Remove a File

Synopsis

#include <stdio.h>

int remove (const char *filename);

Description

Remove causes the file whose name is the string pointed to by *filename* to be removed. Subsequent attempts to open the file will fail, unless it is created anew. If the file is open, the behavior of the *remove* function is the same as *unlink*. *Remove* is implemented as a macro and as a function. To use the function instead of the macro, **#undef** the macro before function invocation.

Return Value

Remove returns zero if the operation succeeds, non-zero if it fails.

See Also

fopen, **open**, **unlink**.

scanf, fscanf, sscanf

(standard I/O library function)

Formatted Input from Stream

Synopsis

#include <stdio.h>

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

int fscanf (FILE *stream, const char *format, ...);

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

Description

Scanf reads from the standard input stream *stdin*. *Fscanf* reads from the named input *stream*. *Sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1 White-space characters (blanks, tabs, new-lines, or form-feeds) which cause input to be read up to the next non-white-space character. White-space in the format string does not mean that white space *must* appear in the input.

2 An ordinary character (not **%**), which must match the next character of the input stream.

3 Conversion specifications, consisting of the character **%**, an optional assignment suppressing character *****, an optional numerical maximum field width, an optional **l** (ell), **L**, or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *****. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except "[" and "c", white space leading an input field is skipped.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

% A single **%** is expected in the input at this point; no assignment is done.

d A decimal integer is expected; the corresponding argument should be an integer pointer.

i A signed integer is expected (whose format is the same as expected by *strtol* when its *base* argument is zero); the corresponding argument should be an integer pointer.

u An unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

o An octal integer is expected; the corresponding argument should be an integer pointer.

x A hexadecimal integer is expected; the corresponding argument should be an integer pointer.

e,f,g A floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a **float**. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optional + or – followed by an integer.

s A character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating **\0**, which will be added automatically. The input field is terminated by a white-space character. Note that *scanf* cannot read a null string.

c

A character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[

Indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the *scanset*, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first-last*, thus [0123456789] may be expressed [0-9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating **\0**, which will be added automatically. At least one character must match for this conversion to be considered successful.

p

A hexadecimal number, which should be the same as the set of sequences that may be produced by the **%p** conversion of the *printf* function. The corresponding argument should be a pointer to a pointer-to-**void**. For any input item other than a value converted earlier during the same program execution, the behavior of **%p** is undefined.

n No input is consumed. The corresponding argument should be a pointer to integer into which is to be written the number of characters read from the input stream so far by this call to the *scanf* function. Execution of an **%n** directive does not increment the assignment count returned at the completion of execution of the *scanf* function.

The conversion characters **d**, **u**, **o**, and **x** may be preceded by **l** or **h** to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** may be preceded by **l** or **L** to indicate that a pointer to **double** or **long double** rather than to **float** is in the argument list (**long double** is equivalent to **double** with this compiler). The **l**, **h**, or **L** modifier is ignored for other conversion characters.

Scanf conversion terminates at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

Scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, **EOF** is returned.

Examples

The call:

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

with the input line:

```
25 54.32E-1 thompson
```

will assign to *n* the value **3**, to *i* the value **25**, to *x* the value **5.432**, and *name* will contain **thompson\0**. Or:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0-9]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* (see **getc**) will return **a**.

See Also

getc, **printf**, **strtod**, **strtol**.

Note

Trailing white space (including a new-line) is left unread unless matched in the control string.

Diagnostics

These functions return **EOF** if an input failure occurs before any conversion. Otherwise, the number of input items assigned (which may be fewer than provided for, or even zero, in case of an early conflict) is returned.

setbuf, setvbuf

Assign Buffering to a Stream File

Synopsis

#include <stdio.h>

void setbuf (FILE *stream, char *buf);

int setvbuf (
FILE *stream,
char *buf,
int type,
size_t size);

Description

Setbuf may be used after a stream has been opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer input/output will be completely unbuffered. A constant **BUFSIZ**, defined in the **<stdio.h>** header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

Setvbuf may be used after a stream has been opened but before it is read or written. *Type* determines how *stream* will be buffered. Legal values for *type* (defined in stdio.h) are:

_IOFBF	Causes input/output to be fully buffered.
_IOLBF	Causes output to be line buffered. The buffer will be flushed when a newline is written, the buffer is full, or when input is requested from other streams.
_IONBF	Causes input/output to be completely unbuffered.

If *buf* is not the **NULL** pointer, the array it points to will be used for buffering instead of an automatically allocated buffer (from **malloc**). *Size* specifies the size of the buffer to be used. The constant **BUFSIZ** in **<stdio.h>** is suggested as a good buffer size. If input/output is unbuffered, *buf* and *size* are ignored.

By default, all input/output is fully buffered.

See Also

fopen, **getc**, **malloc**, **putc**.

Diagnostics

If an illegal value for *type* or *size* is provided, *setvbuf* returns a non-zero value. Otherwise, the value returned will be zero.

Note

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

setjmp, longjmp

Non-Local Goto

Synopsis

#include <setjmp.h>

int setjmp (jmp_buf env);

void longjmp (jmp_buf env, int val);

Description

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* (whose type, *jmp_buf*, is defined in the *<setjmp.h>* header file) for later use by *longjmp*. It returns the value 0.

Longjmp restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed, program execution continues as if the corresponding call of *setjmp* had just returned the value *val*. *Longjmp* cannot cause *setjmp* to return the value 0. If *longjmp* is invoked with a second argument of 0, *setjmp* will return 1.

All globally accessible objects have values as of the time *longjmp* was called. All automatics local to the destination stack frame have values as of the time *setjmp* was called, provided none were modified after calling *setjmp*; if modified, the value of an automatic is undefined.

If a *longjmp* is executed and the environment in which the *setjmp* was executed no longer exists, errors can occur. The conditions under which the environment of the *setjmp* no longer exists include: exiting the procedure which contains the *setjmp* call, and exiting an inner block with temporary storage (e.g., a block with declarations in C, a *with* statement in Pascal). This condition may or may not be detectable. An attempt is made by determining if the stack frame pointer in *env* points to a location not in the currently active stack. If this is the case, *longjmp* will return a –1. Otherwise, the *longjmp* will occur, and if the environment no longer exists, the contents of the temporary storage of an inner block are unpredictable. This condition may also cause unexpected process termination. If the procedure has been exited the results are unpredictable.

Passing *longjmp* a pointer to a buffer not created by *setjmp*, or a buffer that has been modified by the user, can cause all the problems listed above, and more.

Warning

If *longjmp* is called even though *env* was never primed by a call to *setjmp*, or when the last such call was in a function which has since returned, absolute chaos is guaranteed.

setlocale

Locale Control

Synopsis

#include <locale.h>

char *setlocale (int category, const char *locale);

Description

Setlocale selects the appropriate piece of the program's locale as specified by the *category* and *locale* arguments. *Setlocale* may be used to read or modify all or part of the program's current locale. Using **LC_ALL** for *category* specifies the program's entire locale. Other values for *category* name only a part of the program's locale. **LC_COLLATE** affects the behavior of the *strcoll* function. **LC_TYPE** affects the behavior of the character handling functions. **LC_NUMERIC** affects the decimal-point character for the formatted input/output functions (*printf*, *scanf*, etc.) and the string conversion functions (*strtod*, *strtol*, etc.).

A value of "C" for *locale* specifies the minimal environment for C translation; a value of " " for *locale* is equivalent to "C". At present, the only locale that is implemented is "C".

At program startup, the equivalent of

```
setlocale ( LC_ALL, "C" );
```

is executed.

Diagnostics

If a pointer to a string is given for *locale* and the selection can be honored, the *setlocale* function returns the string associated with the specified *category* for the new *locale*. If the selection cannot be honored, the *setlocale* function returns a null pointer, and the program's locale is not changed.

A null pointer for *locale* causes the *setlocale* function to return the string associated with the *category* for the program's current locale; the program's locale is not changed. This inquiry can fail by returning a null pointer only if the *category* is **LC_ALL** and the most recent successful locale-setting call used a *category* other than **LC_ALL**.

The string returned by the *setlocale* function is such that a subsequent call with that string and its associated category will restore that part of the program's locale. The string returned shall not be modified by the program, but may be overwritten by a subsequent call to the *setlocale* function.

See Also

localeconv, **strtod**, **strtol**, **printf**, **scanf**, **strcoll**, **strxfrm**.

sin, cos, tan, asin, acos, atan, atan2

Trigonometric Functions

Synopsis

#include <math.h>

double sin (double x);

double cos (double x);

double tan (double x);

double asin (double x);

double acos (double x);

double atan (double x);

double atan2 (double y, double x);

Description

Sin, *cos* and *tan* return respectively the sine, cosine, and tangent of their argument, *x*, measured in radians.

The approximate limit for the values passed to these functions is 2.98E8 for *sin* and *cos*, and 1.49E8 for *tan*.

Asin returns the arcsine of *x*, in the range –π/2 to π/2.

Acos returns the arccosine of *x*, in the range 0 to π.

Atan returns the arctangent of *x*, in the range –π/2 to π/2.

Atan2 returns the arctangent of *y / x*, in the range –π to π, using the signs of both arguments to determine the quadrant of the return value.

Diagnostics

Sin, *cos*, and *tan* lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return zero when there would otherwise be a complete loss of significance. *errno* is set to **ERANGE**.

If *x* is greater than one for *asin* or *acos*, a Not-a-Number (NaN) is returned. If both arguments for *atan2* are zero, 0.0 is the result. *Errno* is set to **EDOM** for both of these conditions.

Error actions are determined by the bits of a global control flag or the FPU floating-point control register (see the **_fp_error** description).

See Also

_fp_error, "Behavior of Math Library Functions" chapter.

sinh, cosh, tanh

Hyperbolic Functions

Synopsis

```
#include <math.h>

double sinh (double x);

double cosh (double x);

double tanh (double x);
```

Description

Sinh, *cosh*, and *tanh* return, respectively, the hyperbolic sine, cosine, and tangent of their argument. These are double-precision routines.

Diagnostics

Sinh and *cosh* set *errno* to **ERANGE** and return **HUGE_VAL** (*sinh* may return **-HUGE_VAL** for negative *x*) when the correct value would overflow.

These error-handling procedures may be changed with the function **_fp_error**.

See Also

_fp_error, "Behavior of Math Library Functions" chapter.

sqrt

Square Root Function

Synopsis

#include <math.h>

double sqrt (double x);

Description

Sqrt returns the non-negative square root of *x*. The value of *x* may not be negative.

Diagnostics

Sqrt returns a NaN and sets *errno* to **EDOM** when *x* is negative. The error action is determined by the bits of a global control flag or the FPU floating-point control register.

These error-handling procedures may be changed with the function **_fp_error**.

See Also

_fp_error, "Behavior of Math Library Functions" chapter.

strcat, strncat, ...

String Operations

Synopsis

```
#include <string.h>

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

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

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

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

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

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

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

char *strerror (int errnum);

size_t strlen (const char *s);

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

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

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

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

size_t strcspn (const char *s1, const char *s2);

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

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

Description

These functions operate on null-terminated strings. The arguments **s1**, **s2** and **s** point to strings (arrays of characters terminated by a null character). The functions *strcat*, *strncat*, *strcpy*, and *strncpy* all alter **s1**. These functions do not check for overflow of the array pointed to by **s1**.

Strcat appends a copy of string **s2** to the end of string **s1**. *Strncat* appends at most **n** characters. It copies less if *s2* is shorter than *n* characters. Each returns a pointer to the null-terminated result (the original value of *s1*).

Strcmp compares its arguments and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**. *Strncmp* makes the same comparison but looks at most **n** characters (*n* less than or equal to zero yields equality). Both of these routines use **unsigned char** for character comparison.

The *strcoll* function returns an integer greater than, equal to, or less than zero, according to whether the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**. The comparison is based on strings interpreted as appropriate to the program’s locale.

Strcpy copies string **s2** to **s1**, stopping after the null character has been copied. *Strncpy* copies exactly **n** characters, truncating **s2** or adding null characters to **s1** if necessary. The result will not be null-terminated if the length of **s2** is **n** or more. If the length of **s2** is less than **n**, characters from the first null in **s2** to the **n**th character are copied as nulls. Each function returns **s1**.

Note that *strncpy* should not be used to copy *n* bytes of an arbitrary structure. If that structure contains a null byte anywhere, *strncpy* will terminate the copy when it encounters the null byte, thus copying fewer than *n* bytes. Use the *memcpy* function for these cases.

Strerror maps the error number in *errnum* (returned from *errno*) to an error message string. *Strerror* returns a pointer to the string, the contents of which describe the meaning of the error number. The array pointed to must not be modified by the program.

Strlen returns the number of characters in **s**, not including the terminating null character.

Strchr (*strrchr*) returns a pointer to the first (last) occurrence of character **c** (an 8-bit ASCII value) in string **s**, or a NULL pointer if **c** does not occur in the string. The null character terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a NULL pointer if no character from **s2** exists in **s1**.

Strspn (*strcspn*) returns the length of the initial segment of string **s1** which consists entirely of characters from (not from) string **s2**.

Strstr locates the first occurrence in the string pointed to by *s1* of the sequence of characters (excluding the terminating null character) in the string pointed to by *s2*. *Strstr* returns a pointer to the located string, or a null pointer if the string is not found. If the second argument, *s2*, has a length of zero, then *strstr* returns the first argument as the return value.

Strtok considers the string **s1** to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string **s2**. The first call (with pointer **s1** specified) returns a pointer to the first character of the first token, and will have written a null character into **s1** immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string **s1** immediately following that token. In this way subsequent calls will work through the string **s1** until no tokens remain. The separator string **s2** may be different from call to call. When no token remains in **s1**, a NULL pointer is returned.

Since the *strtok* function must keep track of its position in the input string, this function cannot be made reentrant.

Note

For user convenience, all these functions are declared in the optional <*string.h*> header file.

Bugs

The copy operations cannot check for overflow of any receiving string. **NULL** arguments cause undefined behavior.

Character movement is performed differently in different implementations. *Memmove* should be used for overlapping moves.

strtod, atof

String to Double-Precision Number

Synopsis

#include <stdlib.h>

double strtod (const char *str, char **ptr);

double atof (const char *str);

Description

Strtod returns as a double-precision floating-point number the value represented by the character string pointed to by *str*. The string is scanned up to the first unrecognized character.

Strtod recognizes an optional string of "white-space" characters (as defined by *isspace*), then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optional sign, followed by an integer.

If the value of *ptr* is not (char **)NULL, the variable to which it points is set to point at the character after the last number, if any, that was recognized. If no number can be formed, **ptr* is set to *str*, and zero is returned.

Atof(str) is equivalent to *strtod*(str, (char **)NULL).

See Also

scanf, **strtol**.

Diagnostics

If the correct value would cause overflow, **+/-HUGE_VAL** is returned (according to the sign of the value), and *errno* is set to **ERANGE**. If the correct value would cause underflow, zero is returned and *errno* is set to **ERANGE**.

strtol, strtoul, atol, atoi

Convert String to Integer

Synopsis

#include <stdlib.h>

long strtol (const char *str, char **ptr, int base);

unsigned long strtoul (
const char *str,
char **ptr,
int base);

long atol (const char *str);

int atoi (const char *str);

Description

Strtol returns as a long integer the value represented by the character string pointed to by *str*. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters (as defined by *isspace* in **ctype.h**) are ignored.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to *str*, and zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base as follows: After an optional leading sign a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Strtoul is the same as *strtol* except that no leading plus or minus is allowed in the string pointed to by *str*.

Atol(str) is equivalent to *strtol*(str, (char **)NULL, 10).

Atoi(str) is equivalent to (int) *strtol*(str, (char **)NULL, 10).

See Also

atof, **ctype**, **scanf**, **strtod**.

Bugs

Overflow conditions are ignored.

tableS, tableSN, tableU, tableUN

68332 Table Lookup

Synopsis

#include <m68332.h>

int tableS (void *array, short x);
int tableSN (void *array, short x);
unsigned int tableU (void *array, short x);
unsigned int tableUN (void *array, short x);

Description

These functions allow the efficient use of piecewise linear, compressed data tables to model complex functions. The functions are expanded to the 68000 table lookup and interpolate instruction as follows:

tableS is expanded to TBLS (signed, rounded lookup).

tableSN is expanded to TBLSN (signed, unrounded lookup).

tableU is expanded to TBLU (unsigned, rounded lookup).

tableUN is expanded to TBLUN (unsigned, unrounded lookup).

The table may have up to 256 elements. The table element size may be 1, 2, or 4 bytes.

Byte *x*[15:8] is used as an index to the table; *x*[7:0] is used as an interpolation fraction. Rounded results are determined as follows:

array[n] + ((*array*[n+1] – *array*[n]) * *x*[7:0]) / 256

and unrounded results are determined by:

array[n]*256 + ((*array*[n+1] – *array*[n]) * *x*[7:0])

where n is *x*[15:8].

The file **m68332.h** must be included for the table functions to be expanded in-line.

See Also

interpolate, also your assembly language manual.

toupper, tolower, _toupper, _tolower

Translate Characters

Synopsis

#include <ctype.h>

int toupper (int c);

int tolower (int c);

int _toupper (int c);

int _tolower (int c);

Description

Toupper and *tolower* have as domain the range of **getc**: the integers from –1 through 255. If the argument of *toupper* represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged. *Toupper* and *tolower* are implemented both as macros and functions. To use a function instead of a macro, **#undef** the macro before function invocation.

The macros *_toupper* and *_tolower* accomplish the same thing as *toupper* and *tolower* but have restricted domains and are faster. *_toupper* requires a lower-case letter as its argument; its result is the corresponding upper-case letter. The macro *_tolower* requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results. Use of this form will never work with foreign character sets.

See Also

getc.

ungetc

Push Character Back into Input Stream

Synopsis

#include <stdio.h>

int ungetc (int c, FILE *stream);

Description

Ungetc inserts the character *c* into the buffer associated with an input *stream*. That character, *c*, will be returned by the next **getc** call on that *stream*. *Ungetc* returns *c*, and leaves the file *stream* unchanged.

One character of pushback is guaranteed, provided something has already been read from the stream and the stream is actually buffered. In the case that *stream* is *stdin*, one character may be pushed back onto the buffer without a previous read statement.

If *c* equals **EOF**, *ungetc* does nothing to the buffer and returns **EOF**.

Fseek erases all memory of inserted characters.

See Also

fseek, **getc**, **setbuf**.

Diagnostics

Ungetc returns **EOF** if it cannot insert the character.

va_list, va_start, va_arg, va_end

Synopsis

#include <stdarg.h>

va_list
void va_start(va_list list, arg_n)
type va_arg(va_list list, type)
void va_end(va_list list)

Description

The preceding macros are used for functions that have variable numbers of arguments. The type va_list is used to track which of the optional arguments are being processed.

The *va_start* macro is used to initialize the variable of type va_list. Its second argument, arg_n, is the last of the non-optional arguments of the current function. The type of arg_n must be of the default argument promotion types (int, long, double; not char, short, enum, or float).

The *va_arg* macro evaluates to the value of the next optional argument from when the function was invoked. Each successive call to *va_arg* gives the next argument that was given. The second argument to *va_arg* is the type of the argument that was passed next in the list. Again this type should only be from the set of default argument promotion types (int, long, double, pointers, and structures). Using a type of short, char, enum, or float will cause undefined behavior because these types can not be passed as optional arguments.

The *va_end* macro should be called when the last of the optional arguments has been processed. This ensures proper termination of the optional argument processing.

Example

The following function takes a variable number of arguments that are all of type integer. The function returns the sum of all of the optional arguments.

```
#include <stdarg.h>
int
sum(int count, ...)
{
        va_list args;
        int     result = 0;

        va_start(args, count);
        while (count-- > 0)
                result += va_arg(args, int);
        va_end(args);
        return result;
}
```

See also

vprintf

vprintf, vfprintf, vsprintf

Formatted Output of Varargs List

Synopsis

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

int vprintf (const char *format, va_list ap);

int vfprintf (
FILE *stream,
const char *format,
va_list ap);

int vsprintf (
char *s,
const char *format,
va_list ap);
```

Description

Vprintf, *vfprintf*, and *vsprintf* are the same as *printf*, *fprintf*, and *sprintf* respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by **stdargs.h**.

Example

The following demonstrates how *vfprintf* could be used to write an error routine.

```
#include <stdio.h>
#include <stdarg.h>
        . . .
/*        "error" should be called like:
 *        error(function_name, format, arg1, arg2...); */
void
error(char *function_name, char *format, ...)
{
        va_list args;

        va_start(args, format);
        /* print out name of function causing error */
        (void)fprintf(stderr, "ERROR in %s: ", function_name);
        /* print out remainder of message */
        (void)vfprintf(stderr, format, args);
        va_end(args);
        exit(1);
}
```

See Also

printf, **stdarg.h**.

8

Environment-Dependent Routines

Description of the emulator environment-dependent routines.

This chapter describes the HP emulator execution environment-dependent routines. These routines are also used in the debugger/simulator execution environment. The source files for these interface routines (as well as the object code files) are provided so they can be customized for target system execution environments.

The environment-dependent routines (except **monitor** and **mon_stub**) and library functions are all located in linker section name **env**. This section name may be used just as any other section name would be (for example, in **SECTION** pragmas or in the "mode" command line option). See the on-line man pages for a complete description of the cc68k command syntax and options.

The environment-dependent routines relate to the following areas of C programming.

- Program Setup.
- Dynamic Memory Allocation.
- Program Input and Output.

Supported Environments

There are three basic execution environments that these routines support:

- Emulators which use background monitors.
- The debugger/simulator. The simulator does not require a monitor program.
- Emulators which use foreground monitors. Foreground monitors are not supplied for current HP emulators.

Program Setup

Two program setup routines are provided with the C compiler.

crt0.o For programs which use I/O.

crt1.o For programs which do not use I/O.

These routines define the entry point for program setup, **entry()**, and are responsible for general preexecution setup such as initialization of the stack pointer and register A5 (for A5 relative addressing). At the end of preexecution initialization, these setup routines call **main()**.

The source files of the program setup routines have been provided (and are well commented) in case they need to be rewritten, for example, to change any of the default initializations or to add any new program setup such as establishing values other than zero for **argv** and **argc**. Flowcharts of the **crt0** and **crt1** routines are shown in figures 8-1 and 8-2, respectively.

Differences Between "crt0" and "crt1"

The difference between the two program setup routines is that **crt0** will call the **_startup()** library routine to open the standard input, output, and error files: **stdin**, **stdout**, and **stderr**. The **crt1** routine does not open the standard input, output, and error streams and has been provided to avoid the overhead of loading the **stdio** library for a program which doesn't use it.

When using **crt1** instead of **crt0**, the behavior of the **exit()** and **_exit()** library routines is different. Since **crt1** is used in non-I/O applications, neither **exit()** nor **_exit()** will flush buffers or close open files. The **exit()** routine simply executes functions which have been logged by the **atexit()** routine, and the **_exit()** routine just calls **_display_message()**.

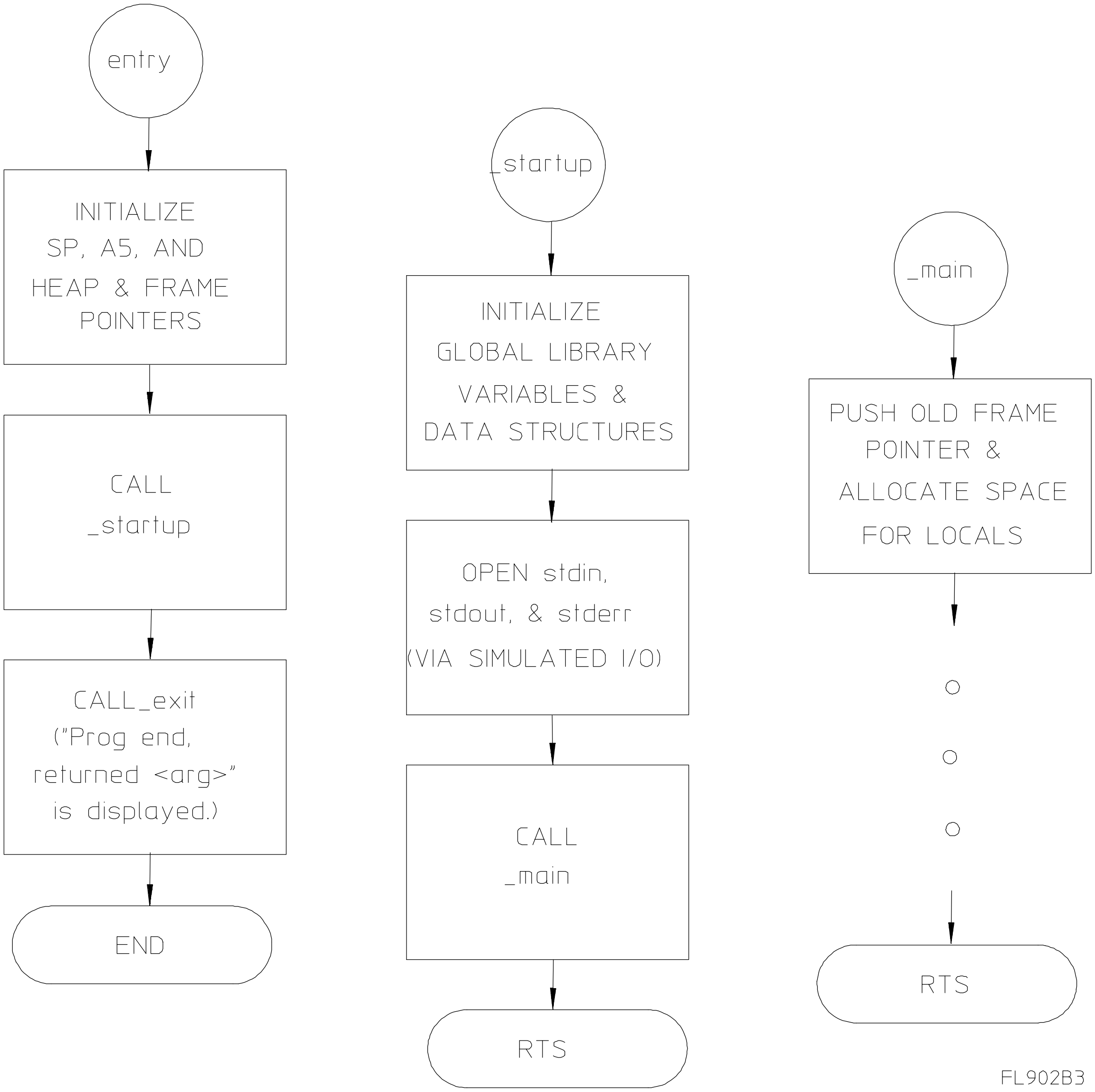

entry
INITIALIZE SP, A5, AND HEAP & FRAME POINTERS
CALL _startup
CALL_exit ("Prog end, returned <arg>" is displayed.)
END
_startup
INITIALIZE GLOBAL LIBRARY VARIABLES & DATA STRUCTURES
OPEN stdin, stdout, & stderr (VIA SIMULATED I/O)
CALL _main
RTS
_main
PUSH OLD FRAME POINTER & ALLOCATE SPACE FOR LOCALS
RTS
FL902B3

Figure 8-1. The "crt0" Program Setup Flowchart

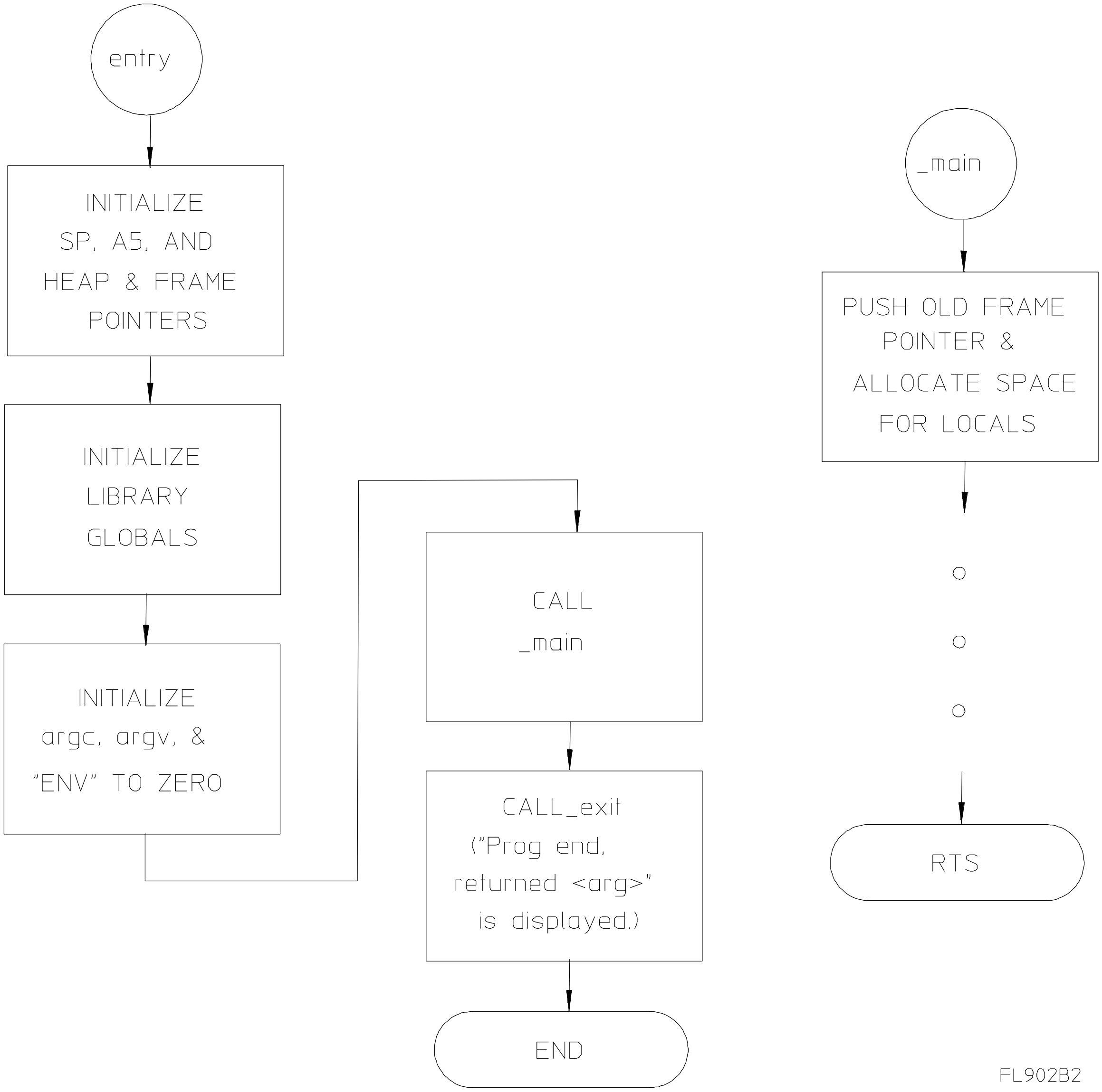

entry
INITIALIZE SP, A5, AND HEAP & FRAME POINTERS
INITIALIZE LIBRARY GLOBALS
INITIALIZE argc, argv, & "ENV" TO ZERO
CALL _main
CALL_exit ("Prog end, returned <arg>" is displayed.)
END
_main
PUSH OLD FRAME POINTER & ALLOCATE SPACE FOR LOCALS
RTS
FL902B2

Figure 8-2. The "crt1" Program Setup Flowchart

The "_display_message()" Routine

The **_display_message()** routine displays run-time error messages. A call to **_display_message()** guarantees program termination. The **_display_message()** routine is called from **_exit()** and other library routines; it is also called by the code generated when the "generate run-time error checking" command line option is specified.

The **_display_message()** routine causes the emulation monitor program to display a message on the emulation display's STATUS line.

An example of how the **_display_message()** routine is called can be found in the **startup.c** source file.

Monitor Considerations

The "mon_stub.s" Routine

The purpose of **mon_stub.s** is to replace the foreground monitor which was used in older HP emulators. The foreground monitor had two purposes: to initialize trap vectors and to provide extra information for the emulator. The **mon_stub.s** routine initializes trap vectors and defines a few global variables which are needed to resolve references in the **_display_message** routine (in source file **disp_msg.s**).

The first 12 exception vectors are defined by **mon_stub.s**. These include zero divide, bus error, illegal instruction, and others. You may customize this routine to use only the exceptions that you are interested in or to change the processing that is done when these exceptions occur. Note that nothing is done with exception 14 (Format error). This exception is specific to the 68010 processor. You may want to add code to handle this exception type if your target processor is the 68010.

The **_display_message** routine is written to work with all version of the HP monitors. Because of this it refers to several identifiers which were formerly defined in the monitor (MONITOR_MESSAGE, JSR_ENTRY). You may wish to rewrite **_display_message** for your environment and remove these definitions.

The emulation monitor stubs shipped with the compiler differ from the source files shipped with the emulators in that the floating-point exception vectors have been initialized with pointers to **fp_traphandler** (contained in library **env.a** and shipped source file **fpu_trap.s**). This allows the floating-point error messages to be more detailed. Another difference is that the integer divide by zero exception vector table entry has been un-commented.

Default Environment Library Setup

The **mon_stub.s** routine is included in the **env.a** library by default. The recommended way for you to add customized exception processing to your system is by changing **mon_stub.s**. If you create a separate source file for your exceptions processing, you must ensure that your code is not linked with **mon_stub.s**; otherwise, there will be section overlap (the vector table is defined in two places).

Linking the Program Setup Routines

The program setup routines are loaded, respectively, by the following linker command files.

iolinkcom.k	Links program with **crt0.o**.
linkcom.k	Links program with **crt1.o**.
fiolinkcom.k	Links program containing 68881/2 code with **crt0.o**.
flinkcom.k	Links program containing 68881/2 code with **crt1.o**.

Since C assumes that **stdin**, **stdout**, and **stderr** are opened prior to **main()** being called, cc68k automatically uses the **iolinkcom.k** (or **fiolinkcom.k**) linker command file. To link with **crt1.o** instead, use the cc68k "no I/O" option to specify that the **linkcom.k** (or **flinkcom.k**) command file be used.

If you use the "generate code for the 68881/2" (**-f**) option, **fiolinkcom.k** or **flinkcom.k** will be used instead of **iolinkcom.k** or **linkcom.k**. These linker command files substitute **lib881.a** for **lib.a** and **libm881.a** for **libm.a**.

Whenever the environment-dependent library, **env.a**, is modified, you must also modify the default linker command file to load the new library.

Emulator Configuration Files

Two configuration files for the emulator are provided:

ioconfig.EA	For programs linked with crt0.o.
config.EA	For programs linked with crt1.o.

Polling for simulated I/O is enabled by the **ioconfig.EA** and **fioconfig.EA** files because the **stdin**, **stdout**, and **stderr** streams (which are set up by the **crt0** routine) are implemented via simulated I/O in the emulation environment. The **config.EA** and **fconfig.EA** files do not enable polling for simulated I/O because **crt1** does not set up the standard input, output, and error streams.

Note that the debugger/simulator does not need these two configuration files.

Default Memory Map

Section ordering is specified in the linker command (**linkcom**) files, and the memory map is specified in the emulator configuration (**config**) files.

Because emulator configuration files map memory for absolute code located by the linker, modifications to the default linker command files will usually require modifications to the emulator configuration files as well.

Dynamic Allocation

There are several dynamic allocation routines in the **libc.a** support library (e.g., **malloc**, **realloc**, etc.). The only environment dependency is isolated in the function **_getmem()**. For these dynamic allocation routines to work, the function **_getmem()** must return memory allocated from the system. The source for the **_getmem()** function is provided in the "shipped sources" directory.

As provided, **_getmem()** returns an address to a block of dynamic memory and the size of that block. If the block size requested by **malloc()** cannot be satisfied, the largest block left in the heap will be returned. The calling sequence is:

```
void *_getmem(int *size);
```

```
ptr = _getmem( &size );
```

The size of the block allocated, whether it is larger or smaller than the size requested, is returned indirectly through the pointer parameter. Calling **_getmem()** with a *size* equal to zero will cause the current address of the heap to be returned.

If desired, **_getmem()** may be written to return more than the requested amount of memory; the dynamic allocation routines will take advantage of this.

Rewriting the "_getmem" Function

This routine should be rewritten to return memory in the best way for the target system. In a simple embedded system this routine should probably be written to return the address of an array big enough to use up all available RAM not used by the rest of the program. If an operating system is present, the routine should be written to return a large chunk of memory from the operating system at each call.

After the **_getmem()** function is rewritten and assembled, use the ar68k librarian to replace the **getmem.o** object module in the **env.a** library.

Input and Output

Many of the functions defined by **stdio.h** use the basic I/O functions found in the **systemio** support library module. These basic I/O functions are: **open()**, **close()**,

read(), **write()**, **lseek()**, and **unlink()**. The **systemio** functions provided use the simulated I/O feature of the emulation environments. The C source code for the basic I/O functions is provided in the "shipped sources" directory.

As provided, the I/O system defines the maximum number of I/O control blocks available as 12 (which equals the maximum number of simulated I/O files that can be open at the same time), and the size of the I/O buffers is defined to be 1020 bytes (based on the 255 byte size of the simulated I/O buffer). These values can be changed by redefining the macros **FOPEN_MAX** and **BUFSIZ** in the header file **stdio.h**; after the values of these macros are changed, you must recompile the file **startup.c**. Changes to **FOPEN_MAX** and **BUFSIZ** will not take effect until a new **startup.o** object file is made and placed in the environment dependent library, **env.a**.

The **systemio.c** file should be rewritten for the target system environment.

After the **systemio.c** file is rewritten and compiled, the new **systemio.o** object file should either be loaded before the **env.a** library, or be used (with ar68k) to replace the existing **systemio** object module in the **env.a** library.

Environment-Dependent I/O Functions

The remainder of this chapter describes the I/O library functions which are dependent on the 68000 emulator execution environment. Functions declared in the **simio.h** include file are found in the environment-dependent library archive file **env.a**.

clear_screen

Clear the Simulated I/O Display

Synopsis

#include <simio.h>

int clear_screen (int fildes);

Description

Clear_screen clears the simulated I/O display if *stdout* is directed to the display. *Fildes* is the file descriptor obtained from an *open* system call to open *stdout*.

Errors

Clear_screen will fail and the display will not be cleared if one of the following conditions is true; *errno* will be set accordingly.

[INVALID_CMD]

Attempt to clear the display on a file that is not a display.

[INVALID_DESC]

Fildes is not an open file descriptor.

[CONTINUE_ERROR]

Attempt to clear the display after a continued emulation session (emulation is exited and then reentered).

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

close

Close a File Descriptor

Synopsis

#include <simio.h>

int close (int fildes);

Description

Fildes is a file descriptor obtained from an *open* system call. *Close* closes the file indicated by *fildes.*

Errors

Close will fail and the file will not be closed if one of the following conditions is true; *errno* will be set accordingly.

[INVALID_DESC]

Fildes is not an open file descriptor.

[CONTINUE_ERROR]

Attempt to close any file descriptor after a continued emulation session (emulation is exited and then reentered).

[UNIX_ERROR]

Any error from the host operating system **close(2)** function.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

See Also

open.

exec_cmd

Execute Operating System Command on the Host

Synopsis

```
#include <simio.h>

int exec_cmd (
const char *command,
int *file1,
int *file2,
int *file3);
```

Description

Exec_cmd executes an operating system command on the host computer. *Command* is a pointer to a string composed of the command to be executed and any parameters required by that command. *File1*, *file2*, and *file3* are pointers to variables which will be set to the file descriptors of the pipes connected to *stdin*, *stdout*, and *stderr* of the process spawned. If any pointer is NULL, that pipe is connected to **/dev/null** and no file descriptor is returned.

Errors

Exec_cmd will fail and the command will not be executed if one of the following conditions is true; *errno* will be set accordingly.

[CANNOT_READ_MEMORY]

Read of command name failed.

[NO_FREE_DESC]

The simulated I/O descriptor table is full.

[TOO_MANY_FILES]

Host **pipe(2)** command failed.

[NO_FREE_PROC_ID]

The maximum number of processes are already active.

[TOO_MANY_PROCESSES]

Host **fork(2)** failed and *errno* = EAGAIN.

[INVALID_CMD_NAME]

The command name length is zero.

[UNIX_ERROR]

Host **fork(2)** failed and *errno* does not equal EAGAIN.

Return Value

Upon successful completion, a process ID number >= 0, and the pipes' file descriptors are returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

exit, _exit

Terminate Process

Synopsis

#include <stdlib.h>

void exit (int status);

void _exit (int status);

Description

Exit is equivalent to *_exit*, except that *exit* flushes stdio buffers, while *_exit* does not. Also, *exit* executes any routines that have been logged by the **atexit** routine; *_exit* does not do this. Both *exit* and *_exit* terminate the calling process by closing all open file descriptors. *_display_message()* is called with the message: "Prog end, returned <arg>", where "arg" is either the value returned by **main()** or the argument passed to an explicit call to *exit*.

When programs are not linked with the I/O routines (the "no I/O" command line option is used), the behavior is the same as above except that *exit* does not flush stdio buffers, and neither function closes open file descriptors.

See Also

atexit.

_getmem

Get Block of Memory from System Heap

Synopsis

#include <memory.h>

void *_getmem(int *size);

Description

_getmem is called by the support library dynamic allocation routines (e.g., **malloc**, **realloc**, etc.) and the **sbrk** function. For these functions to work, *_getmem* must return memory allocated from the system.

_getmem returns an address to a block of dynamic memory and the size of that block. If the block size requested by *malloc* cannot be satisfied, the largest block left in the heap will be returned. *Size* can be negative, in which case the amount of allocated space is decreased.

Return Value

The size of the block allocated, whether it is larger or smaller than the size requested, is returned indirectly through the pointer parameter. Calling *_getmem* with a *size* equal to zero will cause the current address of the heap to be returned.

If desired, *_getmem* may be rewritten to return more than the requested amount of memory; the dynamic allocation routines (e.g., **malloc**, **realloc**, etc.) will take advantage of this.

Warnings

Deallocating memory (calling *_getmem* with a negative *size*) without first having allocated the memory will cause unknown results.

Example

An example of how the *_getmem* function is used can be found in the shipped source file **sbrk.c** shown below.

```
#include <memory.h>
#pragma SECTION PROG=env DATA=envdata CONST=env
extern void *_getmem();

void
*sbrk( incr )
int     incr;
{
        void    *ptr;           /* pointer to memory block allocated   */
        char    *tptr;          /* used to zero memory block allocated */
        int     size = incr;

        ptr = _getmem( &size );
        if( size != incr )              /* was request satisfied? */
        {
                size = -size;           /* free block returned by _getmem since */
                _getmem( &size );       /* did not satisfy request.            */
                return (char *)-1;
        }

        /* initialize memory block to be returned to zero */
        for ( tptr = ptr; tptr < (char *)ptr+incr; tptr++ )
                *tptr = 0;
        return ptr;
}
```

See Also

malloc, **free**, **realloc**, **calloc**, **sbrk**.

initsimio

Initialize Simulated I/O

Synopsis

#include <simio.h>

int initsimio (void);

Description

It is not necessary to call the *initsimio* function prior to calling any other functions implemented via simulated I/O; however, doing so will allow you to restart a program, which was stopped with simulated I/O files still open, without any side effects from the previously opened files.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

kill

Kill Simulated I/O Process

Synopsis

#include <simio.h>

int kill (int pid, int sig);

Description

Kill sends signal *sig* to a process running on the host which is identified by the process ID number *pid*.

Errors

Kill will fail and the process will not be killed if one of the following conditions is true; *errno* will be set accordingly.

[NO_PERMISSION]

Host **kill(2)** failed because of a permissions error.

[INVALID_PROC_ID]

The simulated I/O process ID is unused or out of range (the simulated I/O process entry does not exist).

[INVALID_SIGNAL]

Host **kill(2)** failed because *sig* is not a valid signal.

[NO_SUCH_PROCESS]

The host operating system process does not exist.

[UNIX_ERROR]

Host **kill(2)** failed for some other reason.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

lseek

Move Read/Write File Pointer

Synopsis

#include <simio.h>
#include <stdio.h>

long lseek (int fildes, long int offset, int whence);

Description

Fildes is a file descriptor returned from a *open* system call. *Lseek* sets the file pointer associated with *fildes* as follows. (The SEEK_* macros are defined in *<stdio.h>* which must be included.)

If *whence* is **SEEK_SET**, the pointer is set to *offset* bytes. If *whence* is **SEEK_CUR**, the pointer is set to its current location plus *offset.* If *whence* is **SEEK_END**, the pointer is set to the size of the file plus *offset.*

Upon successful completion, the resulting pointer location, as measured in bytes from the beginning of the file, is returned.

Lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

[INVALID_DESC]

Fildes is not an open file descriptor.

[NO_SEEK_ON_PIPE]

Fildes is associated with a pipe or fifo.

[INVALID_OPTIONS]

Whence is any illegal value.

[INVALID_OPTIONS]

The resulting file pointer would be negative.

[INVALID_CMD]

Fildes is display or keyboard.

[CONTINUE_ERROR]

Attempt to move a file pointer after a continued emulation session (emulation is exited and then reentered).

[UNIX_ERROR]

Some host operating system call has failed. Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

Return Value

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

See Also

open.

open

Open File for Reading or Writing

Synopsis

#include <simio.h>

int open (const char *path, int option);

Description

Open requests the host to open a file specified by *path* with the given *options*. If the operation is successful, *open* will return a valid file descriptor. If unsuccessful, *open* will set *errno* and return -1. *Option* values are constructed by OR-ing flags from the list below.

O_READ	Open for reading only.
O_WRITE	Open for writing only.
O_RDWR	Open for reading and writing.
O_NDELAY	This flag may affect subsequent reads and writes.
O_APPEND	If set, the file pointer will be set to the end of the file prior to each write.
O_CREATE	If the file exists, this flag has no effect. Otherwise, the file is created, the owner ID of the file is set to the effective user ID of the process, and the group ID of the file is set to the effective group ID of the process.
O_TRUNC	If the file exists, its length is truncated to 0 and the mode and owner are unchanged.
O_EXCL	If O_EXCL and O_CREATE are set, *open* will fail if the file exists.

Errors

Open will fail and the file will not be opened if one of the following conditions is true. *Errno* will be set accordingly:

[UNIX_ERROR]

A component of the path prefix is not a directory, or,

The named file is a directory and *option* is write or read/write, or,

The named file resides on a read-only file system and *option* is write or read/write, or,

The named file is a character special or block special file, and the device associated with this special file does not exist, or,

The file is open for execution and *option* is write or read/write. Normal executable files are only open for a short time when they start execution. Other executable file types may be kept open for a long time, or indefinitely under some circumstances, or,

A signal was caught during the *open* system call, or,
The system file table is full.

[FILE_NOT_FOUND]

O_CREATE is not set and the named file does not exist.

[NO_PERMISSION]

A component of the path prefix denies search permission, or,

Option permission is denied for the named file.

[TOO_MANY_FILES]

More than the maximum number of file descriptors are currently open.

[FILE_EXISTS]

O_CREATE and O_EXCL are set, and the named file exists.

[INVALID_FILE_NAME]
Path is null.

[INVALID_OPTIONS]
Option specifies both O_WRITE and O_RDWR. Also, undefined bits set in the *option* parameter.

[NO_FREE_DESC]
The maximum number of simulated I/O files are already open.

Return Value

Upon successful completion, the file descriptor is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

See Also

close, **lseek**, **read**, **write**.

pos_cursor

Position Cursor on Simulated I/O Display

Synopsis

#include <simio.h>

int pos_cursor (int fildes, int col, int row);

Description

Pos_cursor positions the cursor to (column, line) on the display if *stdout* is directed to the display.

Errors

Pos_cursor will fail if one of the following conditions is true; *errno* will be set accordingly.

[INVALID_CMD]
Attempt to position the cursor on a file that is not a display.

[INVALID_ROW_OR_COLUMN]

Row is greater than or equal to 50 rows, or *col* is greater than or equal to 80 columns (or the number of columns on the display, whichever is greater).

[INVALID_DESC]
Fildes is not an open file descriptor.

[CONTINUE_ERROR]
Attempt to position the cursor after a continued emulation session (emulation is exited and then reentered).

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

read

Read Input

Synopsis

#include <simio.h>

int read (int fildes, void *buf, int nbyte);

Description

Read requests the host to read *nbytes* from the file specified by *fildes* and place them into *buf*. If the operation is successful, *read* returns the number of bytes read. If unsuccessful, *read* sets *errno* and returns -1.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a device is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

Errors

Read will fail if one of the following conditions is true and errno will be set accordingly:

[INVALID_DESC]

Fildes is not a valid file descriptor open for reading.

[INVALID_CMD]

Attempt to read from the display.

[CONTINUE_ERROR]

Attempt to read anything after a continued emulation session (emulation is exited and then reentered).

[UNIX_ERROR]

Any error from host **read(2)**.

Return Value

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a -1 is returned and *errno* is set to indicate the error.

Note

Although no more than 255 bytes are transferred from the host at one time, there is no practical limit to the number of bytes that can be read per invocation of *read*.

See Also

open.

sbrk

Get Block of Zero-Filled Memory from System Heap

Synopsis

#include <memory.h>

void *sbrk (int increment);

Description

Sbrk is used to get a block of dynamically allocated memory, *increment* bytes in length, from the system heap. The newly allocated space is set to zero. *Increment* can be negative, in which case the amount of allocated space is decreased.

Return Value

Upon successful completion, *sbrk* returns a pointer to the first byte of the memory block requested. Otherwise, a value of -1 is returned.

Warnings

The pointer returned by *sbrk* is not aligned in any manner. Loading or storing words through this pointer could cause alignment problems.

Care should be taken when using *sbrk* in conjunction with calls to the main memory allocator routines (**malloc**, **calloc**, **realloc**, and **free**). All these routines allocate and deallocate data memory from the system heap. Although you should not attempt this, it is possible to deallocate data memory allocated through the main memory allocator functions with a subsequent call to *sbrk*.

See Also

malloc, **free**, **realloc**, **calloc**, **_getmem**.

unlink

Remove Directory Entry

Synopsis

#include <simio.h>

int unlink (const char *path);

Description

Unlink causes the file whose name is pointed to by *path* to be removed; the file remains open, however, and can be accessed until it is closed. Subsequent attempts to open the file will fail, unless it is created anew.

Errors

Unlink will fail if one of the following conditions is true, and *errno* will be set.

[INVALID_FILE_NAME]

A component of the *path* prefix is not a directory.

[FILE_NOT_FOUND]

The named file does not exist, *path* is NULL, or a component of *path* does not exist.

[NO_PERMISSION]

Search permission is denied for a component of the path prefix. Write permission is denied for the directory containing the file to be removed.

[UNIX_ERROR]

The host **unlink(2)** function failed for some reason other than denied permissions.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

See Also

close, **open**.

write

Write on a File

Synopsis

#include <simio.h>

int write (int fildes, const void *buf, int nbyte);

Description

Write requests the host to write *nbyte* bytes from *buf* to the file specified by *fildes*. If the operation is successful, *write* returns the number of bytes written. If unsuccessful, *write* sets *errno* and returns -1.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the device's current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set when the file is opened, the file pointer will be set to the end of the file prior to the first write.

If a *write* requests that more bytes be written than there is room for, only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

Errors

Write will fail and the file pointer will remain unchanged if one of the following conditions is true and errno will be set accordingly:

[INVALID_DESC]

Fildes is not a valid file descriptor open for writing.

[UNIX_ERROR]

The current file position (as set by *lseek*) is less than zero.

[INVALID_COMMAND]

Fildes indicates the keyboard.

[CONTINUE_ERROR]

Attempt to write anything after a continued emulation session (emulation is exited and then reentered).

Write will fail and the file pointer will be updated to reflect the amount of data transferred if one of the following conditions is true and errno will be set accordingly:

[UNIX_ERROR]

An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.

Return Value

Upon successful completion, the number of bytes actually written is returned. Otherwise, **-1** is returned, and *errno* is set to indicate the error.

See Also

lseek, **open**.

9

Compile-Time Errors

Explanations of compile-time error messages.

Errors are problems which prevent a program from compiling successfully. When you see an error message, you must correct the error then compile the program again.

Warnings are possible problems which may cause your program to execute incorrectly. When you see a warning message, you need to decide whether your code is correct. Warnings are listed at the end of this chapter.

The errors and warnings are listed here in alphabetical order.

In addition to the error or warning message, the compiler shows the line of code, the file name, and the line number.

Errors

Address initializer is too large to fit in declared type. This error can occur when an attempt is made to store a pointer in a variable which was declared with too small a size, such as "short" or "char."

Address of automatic variable is not constant.

Assign of ptr to const to ptr to non-const. This error occurs when a pointer to constant is assigned to a pointer to non-constant. For example:

```
ptr_to_non_const = ptr_to_const;
```

This error prevents the inadvertent modification of constant data via pointers. A cast can be used to override this checking.

Assign of ptr to volatile to ptr to non-volatile. This error occurs when a pointer to volatile is assigned to a pointer to non-volatile.

```
ptr_to_non_volatile = ptr_to_volatile;
```

This error prevents optimizations from being inadvertently made where the **volatile** type modifier has said that they shouldn’t. A cast can be used to override this checking.

Bad command line syntax.

Bad constant expression. This means that a non-constant expression has been used in a context where a constant expression is required.

Bad digit in octal constant.

Bad function declarator. This is a syntax error which occurs when the parser is expecting the start of a function definition. It is often followed by many errors due to the parser being out of sync.

Bad integer constant. This error occurs when a non-integral constant is used in a context where an integer constant is required.

Bit field <name> must be integral type.

Bit width of <bit field name> cannot be 0.

Bit width of <bit field name> too large.

Break must be inside looping construct or switch.

Can only initialize first member of a union.

Can’t access array member of non-lvalue structure.

Can’t declare void object <identifier/member name>. The only objects which may be declared with type **void** are functions returning void and pointers to void.

Cannot assign to a constant. This error occurs when a symbol declared with the "const" type modifier is assigned a value.

Cannot have array of functions. Arrays may not have functions as elements, but they may have *pointers* to functions as elements. (*Hint:* use **`typedef`** to declare a type "pointer to function," then declare an array of this type.)

Cannot have array of void. Although you cannot declare an array of void objects, you may declare an array of pointers to void. For example, you may declare **`void *ptr_array[10]`**.

Cannot take address of a bit field. This error occurs when the unary address operator (&) is used on a bit field.

Cannot take address of a register. This error occurs when the unary address operator (&) is used on a variable declared with the **register** storage class specifier.

Cannot take sizeof this type. Sizeof cannot be applied to a function, bit field, a void, or an undimensioned array.

Case statement must be inside switch.

Case values must be integral.

Character string constant exceeds maximum length. The maximum length for character strings is 1023 characters (1024 if the NULL is counted).

Comment terminator '*/' without comment start.

Condition of '?:' must be scalar. The scalar types include the arithmetic types (char, short, int, long, float, double) and pointers.

Constant literal too large. A constant literal has an implied type. If the value is too large for that type, then an error occurs.

Continue must be inside looping construct.

Control expression must be scalar. The scalar types include the arithmetic types (char, short, int, long, float, double) and pointers.

Declaration for nonexistent parameter. This error occurs when a declaration list of formal parameters contains a declaration for a parameter not listed in the function declarator.

Default statement must be inside switch.

Division or modulo by zero. This error occurs when the compiler determines that a constant folding optimization will cause a divide by zero. Use the unary plus (+) operator to prevent the rearrangement of expressions.

Duplicate label <identifier>.

Duplicate structure or union member <name>.

Empty character literal.

Enum constant value not representable as int. All enumeration values must be representable in an int type.

Exceeded automatic variable space. This error occurs when there is too much local storage. The limit is 2^{31}-1 bytes.

Exceeded parameter passing space. This error occurs when there is too much parameter storage. The limit is 2^{31}-1 bytes.

Expression too complex.

Function call has fewer params than prototype.

Function call has more params than prototype.

Function cannot return array.

Function cannot return function.

Function parameter cannot be void.

Goto non-existent label <identifier>.

Illegal cast operands. This error occurs when an expression cannot be converted to the type specified by the cast construct (for example, casting between a data pointer and a float). The cast operator can only be applied to scalar or void types

Illegal character in input. This is usually caused when a control character has been placed in the C source code.

Illegal function name.

Illegal operand types of <operator>. The operand types are incompatible with the operator.

Illegal preprocessor directive in input.

Incompatible array initializer. The initializer given for an array is not compatible with the type of the array elements.

Incompatible initializer. The initializer given is not compatible with the type of the variable being initialized.

Initializer too large for array.

Interrupt routine must return type void.

Left operand of <operator> must be an lvalue. An "lvalue" is an expression to which values can be assigned.

Missing right delimiter on string literal.

Mixed new and old style parameter declarations.

More initializers than structure members.

Multiple defaults in switch.

Must init arithmetic type with arithmetic value. Arithmetic types (char, short, int, long, float, and double) must be initialized with arithmetic values.

Must initialize bit field with integral constant.

Must init pointer with compatible pointer or 0. A compatible pointer is a pointer with the same type or a data pointer with type **(void *)**. (The NULL pointer constant is 0.)

Negative or zero array size.

No digits in hexadecimal constant.

Only high order dimension of array can be empty.

Operand of <operator> cannot be constant.

Operand of <operator> must be an lvalue. An "lvalue" is an expression to which values can be assigned.

Operand of <operator> must be arithmetic. The arithmetic types are: char, short, int, long, float, and double.

Operand of <operator> must be integral. The integral types are: char, short, int, and long.

Operand of <operator> must be scalar. The scalar types include the arithmetic types (char, short, int, long, float, double) and pointers.

Operand of pointer dereference must be a pointer. Something other than a pointer was found immediately following a dereferencing (indirection) operator *****. Check the declaration of the operand to make sure it is a pointer. You may also see this error message if an arithmetic expression is incorrect (remember that ****** is not an arithmetic operator in C).

Operands of '[]' must be a pointer and an integral. This error occurs when the array name and the index are not alternately a pointer and an integral type (char, short, int, long).

Operands of <operator> must be integral. The integral types are: char, short, int, and long.

Operands of <operator> must be scalar. The scalar types include the arithmetic types (char, short, int, long, float, double) and pointers.

Overflow during floating point constant folding. This error occurs when the compiler determines that a constant folding optimization on floating-point values will cause an overflow. Use the unary plus (+) operator to prevent the rearrangement of expressions.

Param expr type not compatible with prototype.

Param list can only appear in definition. An old style declaration of a function so that another function may use it, like

```
extern char foo ();
```

cannot include parameters, as in

```
extern char foo (a, b);
```

Only the function definition may include a parameter list.

Param type of <name> differs from prototype.

Parameter type must have id in function definition.

Parameters not allowed for interrupt routine.

Parser stack overflow. This error occurs when the compiler has reached a syntactic translation limit. This will only occur in extreme cases. The translation limits are listed in the "C Compiler Overview" chapter.

Redeclaration of section/segment for symbol <id>.
This error occurs when the same symbol is declared in two differently named program sections.

Redeclaration of symbol <identifier>. Rename one of the symbols. In some previous versions of the compiler technology, parameter names were ignored in prototype declarations.

Redeclaration of tag <identifier>.

Redeclaration of whether symbol <identifier> is ORGed.
This error occurs when the same symbol is declared in a relocatable program section and in an absolute program section (defined with the SECTION pragma).

Redefinition of function <identifier>.

Repeated case value.

Return expression does not match function type.

Reuse of absolute address for symbol <name>. This error occurs when absolute address section declarations have been given such that address overlaps occur in the assembly code. All symbols located at a particular address must be in the same section (prog, data, or const) and they must all be either defined in the same module or defined externally.

Section ’lib’ can only be referenced by ’all’. The same addressing mode must be used to call run-time library modules throughout a source file. To do this, use **all** for the *refSect* name with the "mode" option. See the "Libraries" chapter in this manual and the on-line man pages for more information.

Static initializer not a representable constant.

Structure can't contain function <member name>. If you want to store a function in a structure, store a *pointer* to the function. For example, **`int (*funcptr)()`** would be a valid structure element.

Structure can't contain undimensioned array <identifier>. You must give a dimension for any array inside a structure; for example, use **`i[10]`** instead of **`i[]`**.

Structure can't contain void <member name>. Structure elements may not be objects of type **void**. However, pointers to **void** are allowed. For example **`void v`** is not allowed in a structure, but **`void *pv`** is allowed.

Structure element reference of non-structure. The identifier in front of the "." was not declared as a structure.

Switch condition must be integral. In **`switch`**(*expression*) the *expression* must return a value of type **int**.

Syntax error. This error is often caused by a missing semicolon on the preceding line.

Type cannot have zero size. This error will occur if the only member of a structure is a bit field whose size is zero.

Type too large. This error occurs when a type's size is greater than 2^{31}-1 bytes.

Undeclared structure member <name>. This error occurs when you attempt to access a structure member which has not been declared.

Undeclared symbol <identifier>.

Underflow during floating point constant folding. This error occurs when the compiler determines that a constant folding optimization on floating-point values will cause an underflow. Use the unary plus (+) operator to prevent the rearrangement of expressions.

Uninitialized definition of undimensioned array. This error occurs when no dimension is specified in an array declaration. The highest order dimension in an array declaration may be empty if the declaration is initialized.

Unknown or incorrect pragma (ignored).

Unknown type size. This error can occur when a variable declared with the type of an undeclared structure tag is used before the structure is declared.

Unresolved static function <name>. This error indicates that a static function of the form "static f();" was declared, but the function body was never defined.

Warnings

Alias symbol <name> already referenced. Place the **#pragma ALIAS** before the symbol is used. For example, place it immediately before or after the declaration. The alias will not cause substitution of the symbol name in any references which precede the alias.

Array index out of range.

Assignment between different pointer types.

Assignment between pointer and integer.

Cast from less to more restrictive pointer. This warning message is enabled when the cc68k "generate additional warnings" option is specified.

Comparison between different pointer types.

Comparison between pointer and integer.

Confusing line directives may affect debug info. This warning indicates that the line synchronization information passed to the compiler did not correspond to a proper nesting of include files. This is probably due to inconsistent #line directives in the source.

Duplicate const qualifier on type. The type was already declared as const.

Duplicate volatile qualifier on type. The type was already declared as volatile.

Empty body of control statement. This warning message is enabled when the cc68k "generate additional warnings" option is specified.

Empty external declaration.

Extern <identifier> assumed to be in UDATA. The compiler cannot determine if the external identifier was initialized and has placed the identifier in the **UDATA** section. If the variable is initialized, it is very important to place the variable in the correct section (**idata**). To do this, use a **#pragma SECTION DATA=idata** before the external declaration to name the initialized data section. See the "Embedded Systems" chapter for more information. (This condition occurs only when the "separate initialized and uninitialized data" option is used).

External symbol <identifier> exceeds significant length.

Illegal escaped character. Backslash ignored. As an example, the string "\q" would cause the warning to be generated, and the string would become "q".

Local variable <identifier> referenced only once.

Missing parameter declaration (defaulted to int). This warning message is enabled when the cc68k "generate additional warnings" option is specified.

More than one character in character literal.

No emulation local syms if .c and .A file not in same directory. This warning is generated whenever a path to a source file is specified and the "generate HP 64000 format files" option is used. If you will be using an emulator, compile all sources in the directory where they exist.

Non-constant initializer for constant type variable.

Octal or hex character constant too big (truncated).

Shift by out of range constant value.

Static initializer will not be loaded. This warning is enabled when the "uninitialized data" compiler command line option is specified. It warns that there is no load-time initialization for statics and externals

Struct, union, or enum tag used but not declared. It is possible to declare pointers to structures or unions before they are defined. The C language allows this form of forward referencing. This message means that a forward reference for a tag was seen, but never resolved. This warning message is enabled when the cc68k "generate additional warnings" option is specified.

Test expression is an assignment. This warning message is enabled when the cc68k "generate additional warnings" option is specified.

Unreferenced symbol <identifier>. The symbol was declared but is not used.

10

Run-Time Errors

Explanations of run-time error messages.

There are three basic types of run-time error messages. The largest group is generated by floating-point exceptions. The two smaller groups are debug error messages and startup error messages.

Floating-Point Error Messages

In accordance with the IEEE floating-point standard, trapping on floating-point exceptions may be enabled or disabled. (See the **_fp_error** description in the "Libraries" chapter.) If the trap associated with a specific exception is disabled, an IEEE defined value is returned, a global exception flag is set, and no error message is displayed. Conversely, if the trap is enabled and an exception is detected, an error message is displayed on the emulation status line and the program terminates. This type of error message is composed as follows:

68881/2 Libraries:

The 68881/2 floating-point coprocessor's floating-point control register is initialized by the **_set_fp_control** library routine to cause 68000 exceptions on the floating-point errors detailed below. Also, the startup code shipped with the compiler has the interrupt vector table initialized to use **fp_traphandler** (contained in library **env.a** and in shipped source file **fpu_trap.s**) to display the following message when an exception occurs:

```
fp  <error>:  <address of instruction>
```

Processor Libraries:

The following message is composed by the **_fp_error** library routine.

```
fp  <error>:  <function>
```

Where <*error*> is the type of exception and may be one of the following:

operand error This type of error occurs when an operand is invalid for the operation performed. Examples include:

0 * Infinity.
(+Infinity) + (–Infinity).
0/0 or Infinity/Infinity.
A trapping NaN involved in any operation.
Comparison between NaN and any other value.

overflow This type of error occurs when the result of an operation is too large to be represented in the destination format.

underflow This type of error occurs when the result of an operation is too small to be represented in the destination format. If trapping is disabled, the result will be denormalized.

inexact result This type of error occurs when the result requires rounding. Due to the high probability of rounding, this trap is typically disabled.

divide by zero This type of error occurs when attempting to divide a non-zero value by zero. (Zero divided by zero is a special case as an operand error.)

all precision lost This type of error occurs when arguments are reduced and precision is lost.

signaling NaN This type of error occurs if an operand is a signalling (or trapping) NaN.

Where <*function*> may be either a run-time or math function.

Debug Error Messages

If programs are compiled using the "generate run-time error checking" option, code is generated to perform checks for the dereferencing of NULL and uninitialized pointers, and for range errors in array accesses. If one of these conditions occurs, the following type of message is displayed:

Pointer Faults:

```
<file>:<line number>:nil ptr
<file>:<line number>:uninit ptr
```

Range Faults:

```
<file>:<line number>  <index>  >  <max index>
<file>:<line number>  <index>  <  0
```

Where <*file*> refers to the C source file containing the offending instruction. This field is at most 12 characters long and the ".c" extension is removed from the file name.

Where <*line number*> is the line number within the C source file which contains the offending instruction.

Where <*index*> is the index into the array.

And where <*max index*> is the upper bound of the array.

Startup Error Messages

If the **crt0** program setup file is linked with the program, the **startup** routine is called to open the, **stdin**, **stdout**, and **stderr** streams. If for any reason one of these files cannot be opened, the following type of message is displayed:

```
Can’t open  <file>,  prog aborted
```

Where <*file*> is either "stdin", "stdout", or "stderr".

At program termination, a message is always displayed. This message is composed within the **_exit** library routine and is:

```
Prog end, returned <arg>
```

Where <*arg*> is either the value returned by **main()** or the argument passed to an explicit call to **exit()**.

11

Run-Time Library Description

Description of the run-time libraries.

Run-time library routines are usually called by compiler generated code; however, they may be called from assembly language programs as well (including embedded assembly code within the C source file).

The routines listed here may in turn call other subroutines; those subroutines are not listed here.

Note These run-time routines may change in future versions of the compiler.

Conversion Routines

dtof

Casts a 64 bit floating point value to a 32 bit floating point value.

Input:	High 32 bits in register D0. Low 32 bits in register D1.
Output:	Register D0.
Registers Destroyed:	A0, A1, D0, D1.

dtoi

Casts a 64 bit floating point value to a 32 bit **signed** integer by truncation.

Input:	High 32 bits in register D0. Low 32 bits in register D1.
Output:	Register D0.
Registers Destroyed:	A0, A1, D0, D1.

dtoui

Casts a 64 bit floating point value to a 32 bit **unsigned** integer by truncation.

Input:	High 32 bits in register D0. Low 32 bits in register D1.
Output:	Register D0.
Registers Destroyed:	A0, A1, D0, D1.

ftod

Casts a 32 bit float to a 64 bit double.

Input:	Register D0.
Output:	Registers D0 **-** D1.
Registers Destroyed:	A0, A1, D0, D1.

ftoi

Casts a 32 bit float to a 32 bit **signed** integer by truncation.

Input:	Register D0.
Output:	Register D0.
Registers Destroyed:	A0, A1, D0, D1.

ftoui

Casts a 32 bit float to a 32 bit **unsigned** integer by truncation.

Input:	Register D0.
Output:	Register D0.
Registers Destroyed:	A0, A1, D0, D1.

itod

Casts a 32 bit **signed** integer to a 64 bit double.

Input:	Register D0.
Output:	Registers D0 **-** D1.
Registers Destroyed:	A0, A1, D0, D1.

uitod

Casts a 32 bit **unsigned** integer to a 64 bit double.

Input:	Register D0.
Output:	Registers D0 **-** D1.
Registers Destroyed:	A0, A1, D0, D1.

itof

Casts a 32 bit **signed** integer to a 32 bit float.

Input:	Register D0.
Output:	Register D0.
Registers Destroyed:	A0, A1, D0, D1.

uitof

Casts a 32 bit **unsigned** integer to a 32 bit float.

Input:	Register D0.
Output:	Register D0.
Registers Destroyed:	A0, A1, D0, D1.

Floating-Point Routines

add32

Adds two 32 bit floating point values, returning a 32 bit floating point value.

Input:	Addend (x) in register D0. Addor (y) in register D1.
Output:	Register D0.
Registers Destroyed:	A0, A1, D0, D1.

add32z

Adds two 32 bit floating point values, returning a 32 bit floating point value.

Input:	Address of addend (x) in register A0. Addor (y) in register D1.
Output:	Indirect through register A0.
Registers Destroyed:	A0, A1, D0, D1.

add64

Adds two 64 bit floating point values, returning a 64 bit floating point value.

Input: High 32 bits of addend (x) in register A0.
Low 32 bits of addend (x) in register A1.
High 32 bits of addor (y) in register D0.
Low 32 bits of addor (y) in register D1.

Output: Registers D0 - D1 and A0 - A1.

Registers Destroyed: A0, A1, D0, D1.

add64p

Adds two 64 bit floating point values, returning a 64 bit floating point value.

Input: Address of addend (x) in register A1.
High 32 bits of addor (y) in register D0.
Low 32 bits of addor (y) in register D1.

Output: Register D0 - D1 and A0 - A1.

Registers Destroyed: A0, A1, D0, D1.

add64pp

Adds two 64 bit floating point values, returning a 64 bit floating point value.

Input: Address of addend (x) in register A1.
Address of addor (y) in register A0.

Output: Register D0 - D1 and A0 - A1.

Registers Destroyed: A0, A1, D0, D1.

add64z

Adds two 64 bit floating point values, returning a 64 bit floating point value.

Input: Address of addend (x) in register A0.
High 32 bits of addor (y) in register D0.
Low 32 bits of addor (y) in register D1.

Output: Register D0 - D1 and indirect through register A0.

Registers Destroyed: A0, A1, D0, D1.

cmp32

Compares two 32 bit floating point values, returning a value of 0 if op1 = op2 or op1 and op2 are unordered, a value of 1 if op1 > op2, and a value of -1 if op1 < op2.

Input: Operand 1 (x) in register D0.
Operand 2 (y) in register D1.

Output: Register D0.

Registers Destroyed: A0, D0, D1.

cmp32r

Compares two 32 bit floating point values, returning a value of 0 if op1 = op2 or op1 and op2 are unordered, a value of 1 if op1 > op2, and a value of -1 if op1 < op2.

Input: Operand 1 (x) in register D1.
Operand 2 (y) in register D0.

Output: Register D0.

Registers Destroyed: A0, D0, D1.

cmp64

Compares two 64 bit floating point values, returning a value of 0 if op1 = op2 or op1 and op2 are unordered, a value of 1 if op1 > op2, and a value of -1 if op1 < op2.

Input: High 32 bits of operand 1 (x) in reg. A0.
Low 32 bits of operand 1 (x) in reg. A1.
High 32 bits of operand 2 (y) in reg. D0.
Low 32 bits of operand 2 (y) in reg. D1.

Output: Register D0.

Registers Destroyed: A0, A1, D0, D1.

cmp64r

Compares two 64 bit floating point values, returning a value of 0 if op1 = op2 or op1 and op2 are unordered, a value of 1 if op1 > op2, and a value of -1 if op1 < op2.

Input: High 32 bits of operand 1 (x) in reg. D0.
Low 32 bits of operand 1 (x) in reg. D1.
High 32 bits of operand 2 (y) in reg. A0.
Low 32 bits of operand 2 (y) in reg. A1.

Output: Register D0.

Registers Destroyed: A0, A1, D0, D1.

div32

Divides a 32 bit floating point value by another 32 bit floating point value, returning a 32 bit floating point value.

Input: Dividend (x) in register D0.
Divisor (y) in register D1.

Output: Register D0.

Registers Destroyed: A0, A1, D0, D1.

div32r

Divides a 32 bit floating point value by another 32 bit floating point value, returning a 32 bit floating point value.

Input:	Dividend (x) in register D1. Divisor (y) in register D0.
Output:	Register D0.
Registers Destroyed:	A0, A1, D0, D1.

div32z

Divides a 32 bit floating point value by another 32 bit floating point value, returning a 32 bit floating point value.

Input:	Address of dividend (x) in register A0. Divisor (y) in register D1.
Output:	Indirect through register A0.
Registers Destroyed:	A0, A1, D0, D1.

div64

Divides a 64 bit floating point value by another 64 bit floating point value, returning a 64 bit floating point value.

Input:	High 32 bits of dividend (x) in reg. A0. Low 32 bits of dividend (x) in reg. A1. High 32 bits of divisor (y) in register D0. Low 32 bits of divisor (y) in register D1.
Output:	Registers D0 - D1 and A0 - A1.
Registers Destroyed:	A0, A1, D0, D1.

div64p

Divides a 64 bit floating point value by another 64 bit floating point value, returning a 64 bit floating point value.

Input:	Address of dividend (x) in register A1. High 32 bits of divisor (y) in register D0. Low 32 bits of divisor (y) in register D1.
Output:	Registers D0 - D1 and A0 - A1.
Registers Destroyed:	A0, A1, D0, D1.

div64pp

Divides a 64 bit floating point value by another 64 bit floating point value, returning a 64 bit floating point value.

Input:	Address of dividend (x) in register A1. Address of divisor (y) in register A0.
Output:	Registers D0 - D1 and A0 - A1.
Registers Destroyed:	A0, A1, D0, D1.

div64r

Divides a 64 bit floating point value by another 64 bit floating point value, returning a 64 bit floating point value.

Input: High 32 bits of dividend (x) in reg. D0.
Low 32 bits of dividend (x) in reg. D1.
High 32 bits of divisor (y) in register A0.
Low 32 bits of divisor (y) in register A1.

Output: Registers D0 - D1 and A0 - A1.

Registers Destroyed: A0, A1, D0, D1.

div64rp

Divides a 64 bit floating point value by another 64 bit floating point value, returning a 64 bit floating point value.

Input: High 32 bits of dividend (x) in reg. D0.
Low 32 bits of dividend (x) in reg. D1.
Address of divisor (y) in register A1.

Output: Registers D0 - D1 and A0 - A1.

Registers Destroyed: A0, A1, D0, D1.

div64rpp

Divides a 64 bit floating point value by another 64 bit floating point value, returning a 64 bit floating point value.

Input:	Address of dividend (x) in register A0. Address of divisor (y) in register A1.
Output:	Registers D0 - D1 and A0 - A1.
Registers Destroyed:	A0, A1, D0, D1.

div64z

Divides a 64 bit floating point value by another 64 bit floating point value, returning a 64 bit floating point value.

Input:	Address of dividend (x) in register A0. High 32 bits of divisor (y) in register D0. Low 32 bits of divisor (y) in register D1.
Output:	Register D0 - D1 and indirect through register A0.
Registers Destroyed:	A0, A1, D0, D1.

mul32

Multiplies two 32 bit floating point values, returning a 32 bit floating point value.

Input:	Multiplicand (x) in register D0. Multiplier (y) in register D1.
Output:	Register D0.
Registers Destroyed:	A0, A1, D0, D1.

mul32z

Multiplies two 32 bit floating point values, returning a 32 bit floating point value.

Input:	Address of multiplicand (x) in reg. A0. Multiplier (y) in register D1.
Output:	Indirect through register A0.
Registers Destroyed:	A0, A1, D0, D1.

mul64

Multiplies two 64 bit floating point values, returning a 64 bit floating point value.

Input:	High 32 bits of multiplier (x) in register A0. Low 32 bits of multiplier (x) in register A1. High 32 bits of multiplicand (y) in reg. D0. Low 32 bits of multiplicand (y) in reg. D1.
Output:	Registers D0 - D1 and A0 - A1.
Registers Destroyed:	A0, A1, D0, D1.

mul64p

Multiplies two 64 bit floating point values, returning a 64 bit floating point value.

Input:	Address of multiplier (x) in register A1. High 32 bits of multiplicand (y) in reg. D0. Low 32 bits of multiplicand (y) in reg. D1.
Output:	Registers D0 - D1 and A0 - A1.
Registers Destroyed:	A0, A1, D0, D1.

mul64pp

Multiplies two 64 bit floating point values, returning a 64 bit floating point value.

Input: Address of multiplier (x) in register A1.
Address of multiplicand (y) in register A0.

Output: Registers D0 - D1 and A0 - A1.

Registers Destroyed: A0, A1, D0, D1.

mul64z

Multiplies two 64 bit floating point values, returning a 64 bit floating point value.

Input: Address of multiplicand (x) in reg. A0.
High 32 bits of multiplier (y) in reg. D0.
Low 32 bits of multiplier (y) in reg. D1.

Output: Indirect through register A0.

Registers Destroyed: A0, A1, D0, D1.

sub32

Subtracts a 32 bit floating point value from another 32 bit floating point value, returning a 32 bit floating point value.

Input: Minuend (x) in register D0.
Subtrahend (y) in register D1.

Output: Register D0.

Registers Destroyed: A0, A1, D0, D1.

sub32r

Subtracts a 32 bit floating point value from another 32 bit floating point value, returning a 32 bit floating point value.

Input: Minuend (x) in register D1.
Subtrahend (y) in register D0.

Output: Register D0.

Registers Destroyed: A0, A1, D0, D1.

sub32z

Subtracts a 32 bit floating point value from another 32 bit floating point value, returning a 32 bit floating point value.

Input: Address of minuend (x) in register A0.
Subtrahend (y) in register D1.

Output: Indirect through register A0.

Registers Destroyed: A0, A1, D0, D1.

sub64

Subtracts a 64 bit floating point value from another 64 bit floating bit value, returning a 64 bit value.

Input:	High 32 bits of minuend (x) in reg. A0. Low 32 bits of minuend (x) in reg. A1. High 32 bits of subtrahend (y) in reg. D0. Low 32 bits of subtrahend (y) in reg. D1.
Output:	Register D0 - D1 and A0 - A1.
Registers Destroyed:	A0, A1, D0, D1.

sub64p

Subtracts a 64 bit floating point value from another 64 bit floating bit value, returning a 64 bit value.

Input:	Address of minuend (x) in register A1. High 32 bits of subtrahend (y) in reg. D0. Low 32 bits of subtrahend (y) in reg. D1.
Output:	Register D0 - D1 and A0 - A1.
Registers Destroyed:	A0, A1, D0, D1.

sub64pp

Subtracts a 64 bit floating point value from another 64 bit floating bit value, returning a 64 bit value.

Input: Address of minuend (x) in register A1.
Address of subtrahend (y) in reg. A0.

Output: Register D0 - D1 and A0 - A1.

Registers Destroyed: A0, A1, D0, D1.

sub64r

Subtracts a 64 bit floating point value from another 64 bit floating bit value, returning a 64 bit value.

Input: High 32 bits of minuend (x) in reg. D0.
Low 32 bits of minuend (x) in reg. D1.
High 32 bits of subtrahend (y) in reg. A0.
Low 32 bits of subtrahend (y) in reg. A1.

Output: Register D0 - D1 and A0 - A1.

Registers Destroyed: A0, A1, D0, D1.

sub64rp

Subtracts a 64 bit floating point value from another 64 bit floating bit value, returning a 64 bit value.

Input: High 32 bits of of minuend (x) in register D0.
Low 32 bits of minuend (x) in register D1.
Address of subtrahend (y) in register A1.

Output: Register D0 - D1 and A0 - A1.

Registers Destroyed: A0, A1, D0, D1.

sub64rpp

Subtracts a 64 bit floating point value from another 64 bit floating bit value, returning a 64 bit value.

Input: Address of minuend (x) in register A0.
Address of subtrahend (y) in reg. A1.

Output: Register D0 - D1 and A0 - A1.

Registers Destroyed: A0, A1, D0, D1.

sub64z

Subtracts a 64 bit floating point value from another 64 bit floating bit value, returning a 64 bit value.

Input: Address of minuend (x) in register A0.
High 32 bits of subtrahend (y) in reg. D0.
Low 32 bits of subtrahend (y) in reg. D1.

Output: Register D0 - D1 and indirect through register A0.

Registers Destroyed: A0, A1, D0, D1.

Debug Routines

rangefault

Writes signed array index error messsage to status line of emulator.

Input: Address of record in register A0 containing the following:
- Upper bound of array.
- Line number in source where reference occurred.
- Path of source file.

Bad index in register D0.

Output: None.

Registers Destroyed: A0, A1, D0, D1.

rangefaultu

Writes unsigned array index error messsage to status line of emulator.

Input: Address of record in register A0 containing the following:
- Upper bound of array.
- Line number in source where reference occurred.
- Path of source file.

Bad index in register D0.

Output: None.

Registers Destroyed: A0, A1, D0, D1.

ptrfault

Writes a nil/uninitialized pointer error message to status line of emulator.

Input: Address of record in register A0 containing the following:
- Line number in source where dereference occurred.
- Path of source file.

Illegal pointer value in register D1.

Output: None.

Registers Destroyed: A0, A1, D0, D1.

12

Behavior of Math Library Functions

Results of math library functions for various types of floating-point input values.

The first table which follows describes the behavior of the math library functions which are passed a single parameter. The remaining tables describe the math library functions which are passed two parameters.

Wherever the result is an exception, the IEEE defined return value is also listed. The IEEE defined value is returned if trapping on that exception is disabled. (See the **_fp_error** description in the "Libraries" chapter for information on enabling/disabling trapping on floating-point exceptions.)

The 68040 always traps on denormalized numbers. If you expect to encounter denormalized numbers, you should provide code to handle these traps (FP Unimplemented Data Type, vector 55). Trap handling of denormalized numbers may be provided in future versions of the math library.

NUMBER TYPES		**EXCEPTION TYPES**	
D	Denormalized number	DBZ	Divide by zero
N	Normalized number	DMN	Domain error
NaN	Not a number	IOP	Invalid operation
R	Real number	OVR	Overflow
x,y	Function input	RNG	Range error
[ ]	Possible result	TLS	Total loss of significance
		UND	Underflow

Figure 12-1. Legend for Math Library Behavior Tables

Table 12-1. Behavior of Functions with One Parameter

	FUNCTION INPUT								
Funct.	-∞	-N	-D	-0	+0	+D	+N	+∞	NaN
acos	IOP NaN	[IOP NaN]	π/2	π/2	π/2	π/2	[IOP NaN]	IOP NaN	x
asin	IOP NaN	[IOP NaN]	x	0	0	x	[IOP NaN]	IOP NaN	x
atan	-π/2	R	x	0	0	x	R	π/2	x
ceil	-∞	R	0	0	0	1	R	+∞	x
cos	IOP NaN	[TLS NaN]	1	1	1	1	[TLS NaN]	IOP NaN	x
cosh	+∞	[OVR +∞]	1	1	1	1	[OVR +∞]	+∞	x
exp	0	[UND 0.0]	1	1	1	1	[OVR +∞]	+∞	x
floor	-∞	R	-1	0	0	0	R	+∞	x
frexp	IOP NaN	R	R	0	0	R	R	IOP NaN	x
ldexp	-∞	R	R	0	0	R	R	+∞	x
log	IOP NaN	IOP NaN	IOP NaN	IOP -∞	IOP -∞	R	R	+∞	x
log10	IOP NaN	IOP NaN	IOP NaN	IOP -∞	IOP -∞	R	R	+∞	x
modf	IOP NaN	R	R	0	0	R	R	IOP NaN	x
sin	IOP NaN	[TLS NaN]	x	0	0	x	[TLS NaN]	IOP NaN	x
sinh	-∞	[OVR -∞]	1	1	1	1	[OVR +∞]	+∞	x
sqrt	IOP NaN	IOP NaN	IOP NaN	0	0	R	R	+∞	x
tan	IOP NaN	[TLS NaN]	x	0	0	x	[TLS NaN]	IOP NaN	x

Table 12-2. "atan2" Behavior

atan2(x,y)		y								
		-∞	-N	-D	-0	+0	+D	+N	+∞	NaN
x	-∞	IOP NaN	-π/2	-π/2	-π/2	-π/2	-π/2	-π/2	IOP NaN	y
	-N	-π	R	R	-π/2	-π/2	R	R	0	y
	-D	-π	R	R	-π/2	-π/2	R	R	0	y
	-0	-π	-π	-π	IOP 0	IOP 0	0	0	0	y
	+0	π	π	π	IOP 0	IOP 0	0	0	0	y
	+D	π	R	R	π/2	π/2	R	R	0	y
	+N	π	R	R	π/2	π/2	R	R	0	y
	+∞	IOP NaN	π/2	π/2	π/2	π/2	π/2	π/2	IOP NaN	y
	NaN	x	x	x	x	x	x	x	x	x

Table 12-3. "pow" Behavior

pow(x,y)		y								
		-∞	-N	-D	-0	+0	+D	+N	+∞	NaN
x	-∞	0	0	0	1	1	IOP+∞	[IOP+/-∞]	IOP+∞	y
	< -1	0	R	R	1	1	R	R	IOP+∞	y
	= -1	IOP 1.0	R	R	1	1	R	R	IOP 1.0	y
	>-1,<0	IOP +∞	R	R	1	1	R	R	0	y
	-0	IOP NaN	IOP NaN	IOP NaN	IOP NaN	IOP NaN	0	0	0	y
	+0	IOP NaN	IOP NaN	IOP NaN	IOP NaN	IOP NaN	0	0	0	y
	>0,<1	+∞	R	R	1	1	R	R	0	y
	= +1	1.0	R	R	1	1	R	R	1.0	y
	> +1	0	R	R	1	1	R	R	+∞	y
	+∞	0	0	0	1	1	+∞	+∞	+∞	y
	NaN	x	x	x	x	x	x	x	x	x

Table 12-4. "add" Behavior

add(x,y)		y						
		-∞	-N	-0	+0	+N	+∞	NaN
x	-∞	-∞	-∞	-∞	-∞	-∞	IOP NaN	y
	-N	-∞	R	x	x	R	+∞	y
	-0	-∞	y	-0	+0	y	+∞	y
	+0	-∞	y	+0	+0	y	+∞	y
	+N	-∞	R	x	x	R	+∞	y
	+∞	IOP NaN	+∞	+∞	+∞	+∞	+∞	y

Table 12-5. "sub" Behavior

sub(x,y)		y						
		-∞	-N	-0	+0	+N	+∞	NaN
x	-∞	IOP NaN	-∞	-∞	-∞	-∞	-∞	y
	-N	+∞	R	x	x	R	-∞	y
	-0	+∞	-y	+0	-0	-y	-∞	y
	+0	+∞	-y	+0	+0	-y	-∞	y
	+N	+∞	R	x	x	R	-∞	y
	+∞	+∞	+∞	+∞	+∞	+∞	IOP NaN	y
	NaN	x	x	x	x	x	x	x

Table 12-6. "mul" Behavior

mul(x,y)		y						
		-∞	-N	-0	+0	+N	+∞	NaN
x	-∞	+∞	+∞	IOP NaN	IOP NaN	-∞	-∞	y
	-N	+∞	+R	+0	-0	-R	-∞	y
	-0	IOP NaN	+0	+0	-0	-0	IOP NaN	y
	+0	IOP NaN	-0	-0	+0	+0	IOP NaN	y
	+N	-∞	-R	-0	+0	+R	+∞	y
	+∞	-∞	-∞	IOP NaN	IOP NaN	+∞	+∞	y
	NaN	x	x	x	x	x	x	x

Table 12-7. "div" Behavior

div(x,y)		y						
		-∞	-N	-0	+0	+N	+∞	NaN
x	-∞	IOP NaN	+∞	+∞	-∞	-∞	IOP NaN	y
	-N	+0	+R	DBZ +∞	DBZ -∞	-R	-0	y
	-0	+0	+0	IOP NaN	IOP NaN	-0	-0	y
	+0	-0	-0	IOP NaN	IOP NaN	+0	+0	y
	+N	-0	-R	DBZ -∞	DBZ +∞	+R	+0	y
	+∞	IOP NaN	-∞	-∞	+∞	+∞	IOP NaN	y
	NaN	x	x	x	x	x	x	x

Table 12-8. "fmod" and "frem" Behaviors

fmod(x,y)
frem(x,y)
		y						

		-∞	-N	-0	+0	+N	+∞	NaN
x	-∞	IOP NaN	IOP NaN	IOP NaN	IOP NaN	IOP NaN	IOP NaN	y
	-N	x	+R	IOP NaN	IOP NaN	-R	x	y
	-0	-0	-0	IOP NaN	IOP NaN	-0	-0	y
	+0	+0	+0	IOP NaN	IOP NaN	+0	+0	y
	+N	x	+R	IOP NaN	IOP NaN	+R	x	y
	+∞	IOP NaN	IOP NaN	IOP NaN	IOP NaN	IOP NaN	IOP NaN	y
	NaN	x	x	x	x	x	x	x

13

Comparison to C/64000

Information needed to convert files from C/64000.

The Motorola 68000 Family C Cross Compiler is more similar to native C implementations than C/64000. Specifically, it supports register variables as intended by C and it includes a robust set of support libraries.

Another area in which this implementation of C differs significantly from C/64000 is in the area of compiler options. A list of the C/64000 options follows (both general and processor-specific), and comparable options of this implementation are described. Note that many C/64000 options could be specified in the source file and, thus, could be varied within the file; some of the C compiler's comparable options are specified on the command line and affect the entire file.

General C/64000 Options

AMNESIA

This directive in C/64000 encompassed two distinct compiler concerns which are addressed separately in this compiler. First, it was intended to allow for memory mapped I/O locations or locations which could change in value as a result of an asynchronous event such as an interrupt. Second, it was intended to defeat a limited form of common subexpression elimination implemented in C/64000. Both of these intents are addressed by the ANSI standard qualifier **volatile** in this implementation.

ASM_FILE

This is not implemented. A listing with embedded assembly can be provided with the "listing" and "add assembly code to listing" command line options; the "generate assembly source files" option causes assembly source files to be created.

ASMB_SYM

HP format "asmb_sym" files can be generated via a command line option.

DEBUG

This occurs by default. The "strip symbol table information" command line option will remove debug symbols.

EMIT_CODE

This is implemented by a command line option.

END_ORG

This was used to terminate an ORG'd section. In the new compiler, ORG functionality is accomplished via the **SECTION** pragma which is terminated by another **SECTION** pragma.

ENTRY

This is handled by the **crt0** or **crt1** routines to which programs are linked.

EXTENSIONS

This is not supported.

FIXED_PARAMETERS

The intention of this option was to allow the calling of PASCAL/64000 routines from C/64000 routines. This capability can be accomplished through the **ASM** pragma.

FULL_LIST

This is implemented by specifying all the command line options which affect the listing sent to the standard output.

INIT_ZEROS

The main purpose of this option was to avoid large compiler output containing primarily zero initializers for large arrays. This is not a problem with the new assemblers and object file formats which can express large initializers more compactly. There is a related option which gives warnings that no load-time initialization can occur.

LINE_NUMBERS

This occurs by default. The "strip symbol table information" command line option will remove line number symbols.

LIST

This is handled from the command line with the "listing" option.

LIST_CODE

This is handled from the command line with the "listing" option in addition to the "add assembly code to listing" option.

LIST_OBJ

Object listing is always given with "add assembly code to listing" option (specified in addition to the "listing" option).

LONG_NAMES

All internal names in this compiler have 255 character significance; external names have 30 character significance.

OPTIMIZE

This is implemented via the "optimize" command line option.

ORG

This is implemented via the **SECTION** pragma.

PAGE

A page break can be generated by inserting a form feed in the source.

RECURSIVE

This is not implemented since, in C, the user may declare local variables to be static (the only potential gain of this option).

SEPARATE

This option had no effect in the C/64000 C compiler and is not implemented in this compiler. However, the **SECTION** pragma permits control over the sections in which program, data, and constants are placed.

SHORT_ARITH

This is not implemented. However, the new C is able to perform arithmetic calculations on floats without expanding to double which provides much of the savings that this option provided.

STANDARD

This is not implemented.

TITLE

This is not supported.

UPPER_KEYS

This is not supported.

USER_DEFINED

This is not implemented.

WARN

This is implemented via the "suppress warning messages" command line option.

WIDTH

This option caused the 64000/C compiler to read only a portion of a source file line (e.g., the first 80 characters). This option has no equivalent in the C compiler.

68000 Specific C/64000 Options

INTERRUPT

This is implemented in the new C via the INTERRUPT pragma.

TRAP

This option was used in two ways. The first was to declare procedures which were to be entered via TRAP instructions. These could have parameters but no return value and their generated code began with a call to run time library Zenter_trap which copied parameters from the user stack to the system stack prior to executing the procedure. It is anticipated that:

1 Traps are used to execute system functions which are more likely to be written in assembly language.

2 Either parameters are not needed by such system functions or they are implemented as either globals or are passed in a more efficient way than copying them from one stack to another (e.g., in registers).

Therefore, the parameter passing functionality of **TRAP** is not implemented in this compiler. The **INTERRUPT** pragma can be used to produce a C procedure which buffers registers and ends with an RTE.

The second use of this directive was to cause the compiler to generate a TRAP instruction rather than a JSR in calling a function. This may be accomplished (without the parameter passing) by using the **ASM** pragma to embed a TRAP instruction.

BASE_PAGE
FAR
COMMON
CALL_ABS_LONG
CALL_ABS_SHORT
CALL_PC_SHORT
CALL_PC_LONG
LIB_ABS_LONG
LIB_ABS_SHORT
LIB_PC_SHORT
LIB_PC_LONG

These are implemented via the **SECTION** pragma and the "specify addressing mode" command line option.

Differences from HP 64819 Code

This section describes:

1 The differences between the HP 64819 and HP B3640 C compilers.

2 Ways to convert code written for the HP 64819 so that it will work with the B3640 C compiler.

Alignment

HP 64819	Word alignment is set by the $ALIGN ON$ option.
HP B3640	Word alignment is performed. Refer to the "Alignment Considerations" section in the "Internal Data Representations" chapter.

Integral promotions

HP 64819	A **char**, a **short int**, or an **int** bit-field, when used in an expression will be converted to an **int** unless $SHORT_ARITH ON$ is specified.
HP B3640	The effect is the same as if integral promotions were always performed.

Float promotions

HP 64819	Promotion from a **float** to a **double** will be performed in an arithmetic operation unless $SHORT_ARITH ON$ is specified.
HP B3640	Promotion from a **float** to a **double** will not be performed unless one of the operands is a **double**.

Shift operations

HP 64819

Logical shift on all shift operations. Shift by a negative value will reverse the shift direction.

HP B3640

Logical shift on all left shifts and on right shifts of unsigned expressions. Arithmetic shift is used on all right shifts of a signed expression. Shift by a negative value will cause unexpected behavior.

To convert:

Reverse the direction for every negative shift. Cast the expression to unsigned before the shift operation if logical shift is required.

Operations on structures

HP 64819

Structures may be assigned, compared for equality, passed as parameters, or returned from functions.

HP B3640

Structures may be assigned, passed as parameters, and returned from functions. No comparison for equality is allowed.

To convert:

Comparison for equality between structures must be done with in-line code or with user supplied function calls.

Symbol names

HP 64819

The first 15 characters in a symbol name are significant.

HP B3640

Internal names have 255 significant characters.
External names have 30 significant characters.

To convert:

A23456789012345__bcd and A23456789012345__xyz are taken as two different symbols in HP B3640.

Numeric constant formats

HP 64819
$EXTENSIONS ON$ permits use of HP 64000 format for defining binary, octal, decimal, and hexadecimal constants (e.g., 0FFH).

HP B3640
Supports the standard constant formats (e.g., 0xff).

To convert:
Conversion from HP 64000 format to C constant format (e.g., 0FFH to 0xff) is needed.

String constant allocation

HP 64819
Identical string constants or string constants that are a subset of another will be mapped into the same location to minimize space.

HP B3640
Each string constant will have its own memory space allocated in segment **const**.

To convert:
Affects only the assembly code that accesses the absolute location of the constant.

Memory management

HP 64819
INITHEAP, INCREASEHEAP, NEW, DISPOSE, MARK and RELEASE are provided for dynamic memory management.

HP B3640
calloc(), *free()*, *malloc()*, *realloc()*, *__getmem()*, and others are provided.

To convert:
Calls to INITHEAP, NEW, DISPOSE must be converted to calls to *malloc()*, and *free()*. Be aware that the calling sequences and the return values are different in these sets of functions. The heap is initialized during the provided program setup procedures for later use by *__getmem()*.

Math functions

HP 64819

ABS, SQRT, SIN, COS, ARCTAN, LN, and EXP are provided.

HP B3640

abs(), sqrt(), sin(), cos(), atan(), log(), exp(), and others are provided in the standard C arithmetic library.

To convert:

Calls to ABS, SQRT, SIN, COS, ARCTAN, LN, and EXP must be converted to calls to the corresponding function in the C math library.

Passing a byte-sized parameter

HP 64819

All signed and unsigned scalar values are extended to a 16-bit value and then pushed on the stack.

HP B3640

All signed and unsigned scalar values are extended to a 32-bit value and then pushed on the stack. (This is a consequence of 32-bit,rather than 16-bit, integers.)

Passing a pointer

HP 64819

Pointers are pushed on the stack as 32-bit quantities.

HP B3640

Same as HP 64819.

Passing a floating-point value

HP 64819

The floating point value is extended to a 64-bit double precision quantity. The address of this 64-bit value is pushed on the stack. The called routine will copy the value into its local stack area when the function is entered.

HP B3640

All floating point values are pushed on the stack as 64 bit double precision qualities, with the least significant bytes in lower memory addresses.

Passing a structure

HP 64819 Structures less than or equal to 4 bytes are pushed on the stack. Structures greater than 4 bytes will have the addresses pushed and the content of a structure is copied by the called function.

HP B3640 Structures are pushed on the stack on word boundaries. The last word of the structure is passed first.

Passing an array

HP 64819 The address of the array is pushed on the stack.

HP B3640 Same as HP 64819.

Function return values

HP 64819 Values less that or equal to 4 bytes are returned in register D7. Values greater than 4 bytes are stored at the result address pushed by the calling routine.

HP B3640 Values less than or equal to 8 bytes are returned in register D0 (and D1 if necessary). Values greater than 8 bytes are stored at the result address pushed by the calling routine. The result address may point to a static memory location, an automatic variable, or temporary space on the stack.

Removing parameters

HP 64819 The calling routine is responsible for removing parameters from the stack.

HP B3640 Same as HP 64819.

Assembly Code Considerations

Stack frame management is different in the HP 64819 and HP B3640 compilers, as you can see by the parameter passing differences listed above.

The assemblers used with each of the compilers are also different. The HP B3641 assembler is used with the HP B3640 compiler.

Refer to the *68000 Family Assembler, Linker, Librarian* manual for a description of the differences between the assemblers.

When converting assembly language routines, it is best to surround the routines with C function headers and tails and embed your assembly language instructions inside **#pragma ASM** and **#pragma END_ASM** directives. You may have to change the instructions which access the parameters and return values, but if you use the compiler generated symbols (SET equal to A6 offsets), you will be protected should anything about the compiler ever change. Refer to the "Compiler Generated Assembly Code" chapter for information about the HP B3640 compiler’s calling conventions.

14

ASCII Character Set

Asc	Dec	Hex	Oct	Chr	Asc	Dec	Hex	Oct	Chr	Asc	Dec	Hex	Oct	Chr
nul	0	00	000	'\0'	+	43	2B	053		V	86	56	126	
soh	1	01	001	'\1'	,	44	2C	054		W	87	57	127	
stx	2	02	002	'\2'	-	45	2D	055		X	88	58	130	
etx	3	03	003	'\3'	.	46	2E	056		Y	89	59	131	
eot	4	04	004	'\4'	/	47	2F	057		Z	90	5A	132	
enq	5	05	005	'\5'	0	48	30	060		[	91	5B	133	
ack	6	06	006	'\6'	1	49	31	061		\	92	5C	134	'\\'
bel	7	07	007	'\7'	2	50	32	062		]	93	5D	135	
bs	8	08	010	'\b'	3	51	33	063		^	94	5E	136	
tab	9	09	011	'\t'	4	52	34	064		_	95	5F	137	
lf	10	0A	012	'\n'	5	53	35	065		‘	96	60	140	
vt	11	0B	013	'\f'	6	54	36	066		a	97	61	141	
ff	12	0C	014	'\r'	7	55	37	067		b	98	62	142	
cr	13	0D	015	'\15'	8	56	38	070		c	99	63	143	
so	14	0E	016	'\16'	9	57	39	071		d	100	64	144	
si	15	0F	017	'\17'	:	58	3A	072		e	101	65	145	
dle	16	10	020	'\20'	;	59	3B	073		f	102	66	146	
dc1	17	11	021	'\21'	<	60	3C	074		g	103	67	147	
dc2	18	12	022	'\22'	=	61	3D	075		h	104	68	150	
dc3	19	13	023	'\23'	>	62	3E	076		i	105	69	151	
dc4	20	14	024	'\24'	?	63	3F	077		j	106	6A	152	
syn	22	16	026	'\26'	A	65	41	101		l	108	6C	154	
etb	23	17	027	'\27'	B	66	42	102		m	109	6D	155	
can	24	18	030	'\30'	C	67	43	103		n	110	6E	156	
em	25	19	031	'\31'	D	68	44	104		o	111	6F	157	
sub	26	1A	032	'\32'	E	69	45	105		p	112	70	160	
esc	27	1B	033	'\33'	F	70	46	106		q	113	71	161	
fs	28	1C	034	'\34'	G	71	47	107		r	114	72	162	
gs	29	1D	035	'\35'	H	72	48	110		s	115	73	163	
rs	30	1E	036	'\36'	I	73	49	111		t	116	74	164	
us	31	1F	037	'\37'	J	74	4A	112		u	117	75	165	
	32	20	040		K	75	4B	113		v	118	76	166	
!	33	21	041		L	76	4C	114		w	119	77	167	
"	34	22	042		M	77	4D	115		x	120	78	170	
#	35	23	043		N	78	4E	116		y	121	79	171	
$	36	24	044		O	79	4F	117		z	122	7A	172	
%	37	25	045		P	80	50	120		{	123	7B	173	
&	38	26	046		Q	81	51	121		|	124	7C	174	
'	39	27	047	'\''	R	82	52	122		}	125	7D	175	
(	40	28	050		S	83	53	123		~	126	7E	176	
)	41	29	051		T	84	54	124		del	127	7F	177	'\177'
*	42	2A	052		U	85	55	125						

15

About this Version

How this version of the compiler differs from previous versions.

Version 4.01

PC Platform Support

The compiler is now available for personal computers running MS-DOS.

Version 4.00

Compilers have been combined

This compiler now generates code for the following Motorola microprocessors:

- 68000
- 68EC000
- 68HC000
- 68HC001
- 68010
- 68302
- 68020
- 68EC020
- 68030
- 68EC030
- 68040
- 68EC040
- 68331
- 68332
- 68340
- CPU32
- 68881/2 floating-point coprocessors

New product number

The product number has been changed to HP B3640.

For the 68000, the old product number was HP 64902 (for HP 300 hosts) and HP B1460 (for Sun and HP 700 hosts).

For the 68020, the old product number was HP 64903 (for HP 300 hosts) and HP B1461 (for Sun and HP 700 hosts).

For the 68030, the old product number was HP 64907 (for HP 300 hosts) and HP B1478 (for Sun HP 700 hosts).

For the 68332, the old product number was HP 64908 (for HP 300 hosts) and HP B1462 (for Sun and HP 700 hosts).

For the 68040, the old product number was HP 64909 (for HP 300 hosts) and HP B1463 (for Sun and HP 700 hosts).

New command-line options

The **-Wo,m** option tells the optimizer to avoid certain optimizations.

The **-K** option enforces strict section information consistency.

New default environments

All of the default environments supplied with the compiler are now HP 64700-series emulators.

PC-relative libraries

Libraries have been added which access both code and data with PC-relative addressing modes.

More floating-point support

Support has been added for some specialized 68881 instructions: fint, fetoxm1, flog2, flognp1, ftentox, ftwotox, fatanh, and fsincos.

Using the correct version of "as68k"

This compiler is *not* compatible with version 1.20 of the assembler *as68k*. The assembler must be version 2.00 or newer. To find out which version of *as68k* is on your system, type the following:

```
what /usr/hp64000/bin/as68k
```

Re-organized manual

The *User's Guide* and *Reference* manuals have been combined and the chapters have been re-organized a bit.

Version 3.50

Behavior of sprintf

The behavior of the sprintf function is undefined if the destination array is also one of the other arguments. For example, the value of **`string1`** is undefined after the following line of code:

```
sprintf (string1, "%s %d", string1,    integer1);
```

This undefined behavior of sprintf is particularly important because the behavior has changed between versions of the compiler.

Bit fields

The code generated for bit fields has been greatly improved.

Formatted printing

The formatted printing functions, such as printf and sprintf, use less stack space. They use 350 fewer bytes than in version 3.40 compilers.

Streams

The ungetc library function can now be used as the first operation on a stream.

Void pointers

Void pointers now may be compared using the relational operators "<", "<=", ">", and ">=".

Implicit casts

There has been a subtle change in implicit casts in expressions to meet the ANSI C standard. If one operand is long int and the other operand is unsigned int, both are converted to unsigned long int. For example, consider the operation ((double)(ui + l)) where ui is of type unsigned int and l is of type long. In version 3.50, the result is of type unsigned long. In previous versions of the compiler, the result would be of type signed long.

qsort function

The qsort function is now reentrant.

The variable **_qsort_buffer** has been removed from the **libc.a** library. In previous versions of the compiler, this variable needed to be initialized in the program startup code. All references to **_qsort_buffer** should be removed.

Environment library modules

Previous versions of the compiler loaded some modules from **env.a** even though those modules were not used. The library has been restructured so that fewer modules will be loaded.

You may need to load the environment library (**env.a**) twice to resolve all external references. The linker command files (for example, /usr/hp64000/env/hp64744/iolinkcom.k) show how this can be done.

Improved performance

The compile speed has been significantly improved.

68040 function return values

Floating point and double function return values are returned in the FP0 register. The FP0 register is part of the 68040’s built-in floating point unit.

Floating point code generated for the 68040 will not work on other processors, such as the 68000, 68020, and 68030, which lack a built-in floating point unit.

New optimizations

Many new optimizations have been added to the compiler. The assembly code optimizer has been improved as well. LINK and UNLK instructions will be removed from small functions where a frame pointer is not needed. The assembly code optimizer is much better at eliminating common subexpressions.

Because of these changes, you may find that you need to use the "optimize for debugging" option (**-G**) more often than with previous versions of the compiler.

Code sharing

You will see greatly reduced code size if you use sprintf or vsprintf and one of the file-oriented printf routines (printf, fprintf, vprintf, or vfprintf). These functions now share much of their code.

The string versions of the printf routines are still reentrant.

__asm ("C_string") function

In addition to the **#pragma ASM/END_ASM** method of embedding assembly code in the C source, the C compiler supports the **__asm ("C_string")** function. (It is not a true function, but is treated syntactically as a function.) **__asm**, which may only appear inside a function body just as any other function call might, outputs one or more lines of assembly to the output compiler-generated assembly code. The two leading underscores are required and are present to conform to ANSI name space requirements.

The assembly language instructions are contained in the *C_string* argument. The compiler does not check the assembly instructions for correctness. It simply passes the instructions to the assembler. The *C_string* argument must contain whitespace and newlines so assembly instructions will conform to the format and syntax required by the HP B3641 Assembler.

The **__asm** function has two advantages over the ASM/ENDASM pragmas: first, it may be used in macro definitions, and second, it is sometimes more expedient for single instructions.

Modifying function entry/exit code

Three new pragmas are available in this release of the compiler. They are **#pragma FUNCTION_ENTRY "C_string"**, **#pragma FUNCTION_EXIT**

"C_string", and **#pragma FUNCTION_RETURN "C_string"**. These pragmas allow you to insert embedded assembly code in the entry and exit code of a function. They are useful for monitoring and debugging function calls.

Index

Certification and Warranty

Certification

Hewlett-Packard Company certifies that this product met its published specifications at the time of shipment from the factory. Hewlett-Packard further certifies that its calibration measurements are traceable to the United States National Bureau of Standards, to the extent allowed by the Bureau's calibration facility, and to the calibration facilities of other International Standards Organization members.

Warranty

This Hewlett-Packard system product is warranted against defects in materials and workmanship for a period of 90 days from date of installation. During the warranty period, HP will, at its option, either repair or replace products which prove to be defective.

Warranty service of this product will be performed at Buyer's facility at no charge within HP service travel areas. Outside HP service travel areas, warranty service will be performed at Buyer's facility only upon HP's prior agreement and Buyer shall pay HP's round trip travel expenses. In all other cases, products must be returned to a service facility designated by HP.

For products returned to HP for warranty service, Buyer shall prepay shipping charges to HP and HP shall pay shipping charges to return the product to Buyer. However, Buyer shall pay all shipping charges, duties, and taxes for products returned to HP from another country. HP warrants that its software and firmware designated by HP for use with an instrument will execute its programming instructions when properly installed on that instrument. HP does not warrant that the operation of the instrument, or software, or firmware will be uninterrupted or error free.

Limitation of Warranty

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse, operation outside of the environment specifications for the product, or improper site preparation or maintenance.

No other warranty is expressed or implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.

Exclusive Remedies

The remedies provided herein are buyer’s sole and exclusive remedies. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory.

Product maintenance agreements and other customer assistance agreements are available for Hewlett-Packard products.

For any assistance, contact your nearest Hewlett-Packard Sales and Service Office.