User’s Guide

HP B1493 8086/186 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.

HP-UX 9.* and 10.0 for HP 9000 Series 700 and 800 computers are X/Open Company UNIX 93 branded products.

MS-DOS is a U.S. registered trademark of Microsoft Corp.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

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

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	64904-90902, September 1988 E0988
Edition 2	64904-97000/97001, September 1989
Edition 3	64904-97002/97003, February 1990
Edition 4	B1493-97000, September 1993
Edition 5	**B1493-97001, June 1995**

B1493-97000 incorporates information which previously appeared in 64904-92007, 64904-97002, and 64904-97003.

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 8086/186 C Cross Compiler translates C source code into 8086/186 assembly language which can be accepted by the HP B1449 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 8087 floating point coprocessor.
- Three ways to embed assembly language in C source.
- Named section specification in C source.
- Choice of small or large memory model 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 2 Compiler Reference

2 C Compilation Overview

3 Internal Data Representation

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 8086/186 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 Intel 8086 microprocessor architecture.
- Basic host operating system commands (such as **cp**, **mv**, **ls**, **mkdir**, **rm**, and **cd**) and a text editor (such as **vi**).

In addition, most sections in this manual assume that you are familiar with 8086/186 assembly language.

Parts of the compiler

The "compiler" is really a set of programs:

- **cc8086**, the C compilation control command.
- **cpp8086**, the C preprocessor.
- **clst8086**, the lister.
- **ccom8086**, the C compiler.
- **opt8086**, the peephole optimizer.

The compiler makes use of several assembler programs:

- **ap86**, the assembler preprocessor.
- **as86**, the assembler.
- **ld86**, the linking loader.

To compile a C program, you can use just the **cc8086** C compilation control command. The **cc8086** command will run the other programs as needed.

Summary of compiler options

-b	Invoke Basis Branch Analyzer preprocessor.
-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 segments.
-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 8087 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 segment consistency.
-l*x*	Search **lib***x***.a** when linking.
-L[**i**][**x**]	Generate ".O" 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 *memoryModel*	Specify memory model, **small**, **compact**, **medium**, or **large**.

-M	Cause generation of more warning messages than are generated by default.
-n	Cause static functions in the large memory model to be called "NEAR".
-N	Cause linking with **linkcom.k** (no I/O) rather than **iolinkcom.k**.
-o *outfile*	Name absolute file *outfile* instead of **a.out.x**.
-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	Byte align data in memory instead of default word alignment.
-r *dir*	Use default linker command file in /usr/hp64000/env/*dir* 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

.a	Library (archive) files.
.A	HP format assembler symbol file.
.c	C source files.
.EA, .EB	Emulator configuration files.
.h	Include (header) files.
.i	"Preprocess only" output (generated with the **-P** option).
.k	Linker command file.
.L	HP format linker symbol file.
.o	Relocatable object file.
.O	Listing files (generated with the **-L** option).
.s	Assembly language source file.
.x	HP-OMF 86 absolute (executable) file.
.X	HP format absolute (executable) file. (Generated with the **-h** option.)
.Ys	Symbol file directory.

To install the software

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 create a simple C program

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

```
main()
{
  char str[80];

  printf("Enter string: ");
  gets(str);
  printf("\nYou entered: \"%s\"\n", str);
}
```

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

To compile a simple program

- Use the cc8086 comand at your host operating system prompt.

Example

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

```
cc8086 simple.c
```

This command generates the executable file **a.out.x** by default. 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:

```
cc8086 -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**.

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 select a memory model

The 8086/186 C compiler allows you to select one of four available memory models: large, medium, compact, or small. The compiler defaults to the large memory model (option **-m large**).

Large Model

The large memory model allows your code and data to be broken up into many named segments of your own choosing. These segments can be located anywhere in memory at link time, independent of each other. Segments containing "constant" data may be located next to code segments to facilitate putting code and data constants in ROM. In fact, "constant" data may be placed in the same segment as code. Segments may also be "ORGed" to absolute physical memory locations through the use of the **# pragma SEGMENT** directive. The efficiency of the compiler in calling functions and accessing data is controllable through the use of **# pragma SEGMENT** and **# pragma DS** directives and the **-n** option.

Small Model

The small memory model produces more compact code than the large memory model. The small memory model places all code in a single, pre-defined, physical segment and places all data, stack, heap, and constants into a second, pre-defined physical segment. Code in the first segment is accessed "segment-relative,", but all data in the second physical segment is accessed "group-relative" because the pre-defined segments that are combined to form the second physical segment are part of a group named **data_const**. Because the small memory model uses just two physical segments, code is limited to 64K bytes and data, stack, heap, and constants together are limited to 64K bytes.

The **# pragma SEGMENT** directive and the **# pragma DS** directive cannot be used with the small memory model and are therefore warned at and ignored if they are encountered. Also, small memory model does not support "ORGing" a segment because this conflicts with the "rules" of small memory model.

Note It is possible to place constant data in ROM when using the small memory model if the embedded environment has RAM near the ROM and both RAM and ROM can be addressed within the 64K limit required for group-relative accesses. If the embedded system cannot meet these requirements, then the constant data must be placed in RAM and initialized at either load-time or at run-time (depending upon the embedded environment).

Medium Model

The medium memory model has one or more code segments, like the large memory model, and one data segment.

Compact Model

The compact memory model has one or more data segments, like the large model, and one code segment.

An Example Using Large Memory Model

The compiler generates code to load the DS register with the paragraph number of the currently active static data segment (providing any such accesses are made in the function) at the beginning of each function. Thereafter, accesses to items in the data segment are performed DS-relative and all other accesses are performed ES-relative; which is far more expensive since ES must, potentially, be reloaded prior to each access. Thus, care should be taken to have the most appropriate data segment active at function definition (using the **# pragma SEGMENT** and **# pragma DS** directives).

Auto variables and parameters are accessed SS-relative since they are on the stack and therefore located in the *userstack* segment.

The example program demonstrates how the compiler selects the segment with which to perform DS-relative accesses. When **largemodel.c** is compiled using the following command line, the **largemodel.O** listing file results.

```
$  cc8086 -SOL  largemodel.c  <RETURN>
```

```
#pragma SEGMENT DATA=my_data1
int i1;
#pragma SEGMENT PROG=my_prog2 DATA=my_data2
int i2;         /* This is in the "active" static data segment. */

void function()
{
  i1 += i2;     /* i2 is in "active" static data segment. */
}

#pragma SEGMENT PROG=my_prog1
main()
{
  long a1;      /* a1 is a dynamic variable on the stack. */

  a1 = 1;       /* Accessed SS-relative.  */
  i1++;         /* Accessed DS-relative. */
  i2++;         /* Accessed ES-relative. */
  function();
}
```

Figure 1-2. largemodel.c

```
HPB1493-19303  8086 C Cross Compiler A.04.01  largemodel.c

;
;MKT:@(#) B1493-19303 A.04.01 8086 C CROSS COMPILER                    03May95
; Memory Model:  large
;
$PAGEWIDTH(230)
$NOPAGING
        NAME    "largemodel"
%DEFINE(MM_CHECK_)(MM_CHECK_L)
%DEFINE(lib)(lib)
%DEFINE(SS)(SS)
%DEFINE(DS)(DS)
%DEFINE(ALIGN)(WORD)
%DEFINE(DALIGN)(WORD)
my_prog2        SEGMENT %ALIGN PUBLIC 'CODE'
%DEFINE(CodeSegment)(my_prog2)
    1   #pragma SEGMENT DATA=my_data1
    2   int i1;
    3   #pragma SEGMENT PROG=my_prog2 DATA=my_data2
    4   int i2;         /* This is in the "active" static data segment. */
    5
    6   void function()
    7   {
        PUBLIC  _function
        ASSUME  CS:%CodeSegment,DS:my_data2
_function       PROC    FAR
%SET(SAVE_ALL_NPX,2)
        PUSH    DS
        MOV     AX,my_data2
        MOV     DS,AX
    8     i1 += i2;     /* i2 is in "active" static data segment. */
        MOV     DX,SEG _i1
        MOV     DI,OFFSET _i1+0
        MOV     AX,%DS:WORD PTR _i2[0]
        MOV     ES,DX
        ADD     ES:WORD PTR [DI],AX
    9   }
functionExit1:
        POP     DS
returnLabel1:
        RET
_function       ENDP
my_prog2        ENDS
my_prog1        SEGMENT %ALIGN PUBLIC 'CODE'
%DEFINE(CodeSegment)(my_prog1)
   10
   11   #pragma SEGMENT PROG=my_prog1
   12   main()
   13   {
        PUBLIC  _main
        ASSUME  CS:%CodeSegment,DS:my_data2
_main   PROC    FAR
%SET(SAVE_ALL_NPX,2)
        PUSH    BP
        MOV     BP,SP
```

Figure 1-3. largemodel.O

```
        SUB     SP,4
        PUSH    DS
        MOV     AX,my_data2
        MOV     DS,AX
%SET(S_a1,-4)
   14     long a1;      /* a1 is a dynamic variable on the stack. */
   15
   16     a1 = 1;       /* Accessed SS-relative.  */
        MOV     SS:WORD PTR [BP+%S_a1+0],1
        MOV     SS:WORD PTR [BP+%S_a1+0+2],0
   17     i1++;         /* Accessed DS1-relative. */
        MOV     DX,SEG _i1
        MOV     DI,OFFSET _i1+0
        MOV     ES,DX
        INC     ES:WORD PTR [DI]
   18     i2++;         /* Accessed DS0-relative. */
        INC     %DS:WORD PTR _i2[0]
   19     function();
        CALL    FAR PTR _function
   20   }
functionExit2:
        POP     DS
        MOV     SP,BP
        POP     BP
returnLabel2:
        RET
_main   ENDP
my_prog1        ENDS
my_data1        SEGMENT %DALIGN PUBLIC
        PUBLIC  _i1
        EVEN
_i1     LABEL   BYTE
        DB      2 DUP(0)
my_data1        ENDS
my_data2        SEGMENT %DALIGN PUBLIC
        PUBLIC  _i2
        EVEN
_i2     LABEL   BYTE
        DB      2 DUP(0)
my_data2        ENDS
        EXTRN   %MM_CHECK_:BYTE
mm_check        SEGMENT BYTE COMMON
        DW      OFFSET %MM_CHECK_
mm_check        ENDS
        END
```

Figure 1-3. largemodel.O (continued)

An Example Using Small Memory Model

Notice that the difference between the source file for large model and the source file for small model is the absence of the **# pragma SEGMENT** directives. The **# pragma SEGMENT** directive and the **# pragma DS** directive are not valid for small model. If they appear, they are warned and ignored.

All functions are called "NEAR" and are accessed CS-relative. Data, stack, heap, and constants all become part of the **data_const** group and are accessed group-relative. The DS, ES, and SS registers are loaded with the same value, the **data_const** paragraph number, by the program startup routine (**crt0.o**).

The smallmodel.O listing shows some of the pre-defined segments that make up the **data_const** group. (Segments **heap** and **userstack** are added to the group at link time.) Through the use of an ASSUME statement, the DS register is associated with the group base of **data_const** instead of a segment base value. For this reason, all DS-relative accesses to data are group-name-relative instead of segment-name-relative.

```
int i1;              /* Will be put in segment "data". */
int i2;         /* Will be put in segment "data". */

void function()
{               /* Will be put in segment "prog/CODE"  */
                /* (segment "prog" is in class "CODE").*/
  i1 += i2;
}

main()
{               /* Will be put in segment "prog/CODE". */
  long a1;      /* a1 is a dynamic variable on the stack. */

  a1 = 1;
  i1++;
  i2++;
  function();
}
```

Figure 1-4. smallmodel.c

When **smallmodel.c** is compiled using the following command line, the **smallmodel.O** listing file results.

```
$  cc8086 -SOL -m small smallmodel.c <RETURN>
```

```
HPB1493-19303  8086 C Cross Compiler A.04.01  smallmodel.c

;
;MKT:@(#) B1493-19303 A.04.01 8086 C CROSS COMPILER                    03May95
; Memory Model:  small
;
$PAGEWIDTH(230)
$NOPAGING
        NAME    "smallmodel"
%DEFINE(MM_CHECK_)(MM_CHECK_S)
%DEFINE(SS)(DS)
%DEFINE(DS)(data_const)
%DEFINE(GRP)(data_const)
%GRP    GROUP data,idata,udata,const
data    SEGMENT WORD PUBLIC
data    ENDS
idata   SEGMENT WORD PUBLIC
idata   ENDS
udata   SEGMENT WORD PUBLIC
udata   ENDS
const   SEGMENT WORD PUBLIC
const   ENDS
prog    SEGMENT BYTE PUBLIC 'CODE'
    1   int i1;         /* Will be put in segment "data". */
    2   int i2;         /* Will be put in segment "data". */
    3
    4   void function()
    5   {               /* Will be put in segment "prog/CODE"  */
        PUBLIC  _function
        ASSUME  CS:prog,DS:%GRP
_function       PROC    NEAR
%SET(SAVE_ALL_NPX,2)
    6                   /* (segment "prog" is in class "CODE").*/
    7     i1 += i2;
        MOV     AX,%DS:WORD PTR _i2[0]
        ADD     %DS:WORD PTR _i1[0],AX
    8   }
functionExit1:
returnLabel1:
        RET
_function       ENDP
    9
   10   main()
   11   {               /* Will be put in segment "prog/CODE". */
        PUBLIC  _main
        ASSUME  CS:prog,DS:%GRP
_main   PROC    NEAR
%SET(SAVE_ALL_NPX,2)
        PUSH    BP
        MOV     BP,SP
```

Figure 1-5. smallmodel.O

```
        SUB     SP,4
%SET(S_a1,-4)
   12     long a1;      /* a1 is a dynamic variable on the stack. */
   13
   14     a1 = 1;
        MOV     SS:WORD PTR [BP+%S_a1+0],1
        MOV     SS:WORD PTR [BP+%S_a1+0+2],0
   15     i1++;
        INC     %DS:WORD PTR _i1[0]
   16     i2++;
        INC     %DS:WORD PTR _i2[0]
   17     function();
        CALL    NEAR PTR _function
   18   }
functionExit2:
        MOV     SP,BP
        POP     BP
returnLabel2:
        RET
_main   ENDP
prog    ENDS
data    SEGMENT WORD PUBLIC
        PUBLIC  _i1
        EVEN
_i1     LABEL   BYTE
        DB      2 DUP(0)
        PUBLIC  _i2
        EVEN
_i2     LABEL   BYTE
        DB      2 DUP(0)
data    ENDS
        EXTRN   %MM_CHECK_:BYTE
mm_check        SEGMENT BYTE COMMON
        DW      OFFSET %MM_CHECK_
mm_check        ENDS
        END
```

Figure 1-5. smallmodel.O (continued)

An Example Using Compact Memory Model

All functions are called "FAR." Data, stack, heap, and constants all become part of the **data_const** group and are accessed group-releative just as with the small memory model.

Through the use of an ASSUME statement, the DS register is associated with the group base of **data_const** instead of a segment base value. For this reason, all DS-relative accesses to data are group-name-relative instead of segment-name-relative.

```
#pragma SEGMENT DATA=my_data1
int i1;
#pragma SEGMENT PROG=my_prog2 DATA=my_data2
int i2;         /* This is in  the "active" static data segment. */

void function()
{
  i1 += i2;     /* i2 is in "active" static data segment. */
}

#pragma SEGMENT DATA=my_prog1
main()
{
  long a1;      /* a1 is a dynamic variable on the stack. */

  a1 = 1;       /* Accessed SS-relative. */
  i1++;         /* Accessed DS-relative. */
  i2++;         /* Accessed DS-relative. */
  function();
}
```

Figure 1-6. compactmodel.c

When **compactmodel.c** is compiled using the following command line, the **compactmodel.O** listing file results.

```
$  cc8086 -SOLm compact compactmodel.c <RETURN>
```

```
HPB1493-19303  8086 C CROSS COMPILER  A.04.01  compactmodel.c

;
;MKT:@(#) B1493-19303 A.04.01 8086 C CROSS COMPILER                    03May95

; Memory Model:  compact
;
$PAGEWIDTH(230)
$NOPAGING
        NAME    "compactmodel"
%DEFINE(MM_CHECK_)(MM_CHECK_C)
%DEFINE(SS)(SS)
%DEFINE(DS)(DS)
%DEFINE(ALIGN)(WORD)
%DEFINE(DALIGN)(WORD)
my_prog2        SEGMENT %ALIGN PUBLIC 'CODE'
%DEFINE(CodeSegment)(my_prog2)
    1   #pragma SEGMENT DATA=my_data1
    2   int i1;
    3   #pragma SEGMENT PROG=my_prog2 DATA=my_data2
    4   int i2;         /* This is in  the "active" static data segment. */
    5
    6   void function()
    7   {
        PUBLIC  _function
        ASSUME  CS:prog,DS:my_data2
_function       PROC    NEAR
%SET(SAVE_ALL_NPX,2)
        PUSH    DS
        MOV     AX,my_data2
        MOV     DS,AX
    8     i1 += i2;     /* i2 is in "active" static data segment. */
        MOV     DX,SEG _i1
        MOV     DI,OFFSET _i1+0
        MOV     AX,%DS:WORD PTR _i2[0]
        MOV     ES,DX
        ADD     ES:WORD PTR [DI],AX
    9   }
functionExit1:
        POP     DS
returnLabel1:
        RET
_function       ENDP
   10
   11   #pragma SEGMENT DATA=my_prog1
   12   main()
   13   {
        PUBLIC  _main
        ASSUME  CS:prog,DS:NOTHING
_main   PROC    NEAR
%SET(SAVE_ALL_NPX,2)
        PUSH    BP
        MOV     BP,SP
        SUB     SP,4
%SET(S_a1,-4)
   14     long a1;      /* a1 is a dynamic variable on the stack. */
   15
```

Figure 1-7. compactmodel.O

```
   16     a1 = 1;       /* Accessed SS-relative. */
        MOV     SS:WORD PTR [BP+%S_a1+0],1
        MOV     SS:WORD PTR [BP+%S_a1+0+2],0
   17     i1++;         /* Accessed DS-relative. */
        MOV     DX,SEG _i1
        MOV     DI,OFFSET _i1+0
        MOV     ES,DX
        INC     ES:WORD PTR [DI]
   18     i2++;         /* Accessed DS-relative. */
        MOV     DX,SEG _i2
        MOV     DI,OFFSET _i2+0
        MOV     ES,DX
        INC     ES:WORD PTR [DI]
   19     function();
        CALL    NEAR PTR _function
   20   }
functionExit2:
        MOV     SP,BP
        POP     BP
returnLabel2:
        RET
_main   ENDP
my_prog2        ENDS
my_data1        SEGMENT %DALIGN PUBLIC
        PUBLIC  _i1
        EVEN
_i1     LABEL   BYTE
        DB      2 DUP(0)
my_data1        ENDS
my_data2        SEGMENT %DALIGN PUBLIC
        PUBLIC  _i2
        EVEN
_i2     LABEL   BYTE
        DB      2 DUP(0)
my_data2        ENDS
        EXTRN   %MM_CHECK_:BYTE
mm_check        SEGMENT BYTE COMMON
        DW      OFFSET %MM_CHECK_
mm_check        ENDS
        END
```

Figure 1-7. compactmodel.O (continued)

An Example Using Medium Memory Model

All functions are called "NEAR" and are accessed CS-relative.

The ES, DS, and SS registers are loaded with the same value, **the data_const** paragraph number, by the program startup routine **crt0.o**, just as with the small memory model.

```
#pragma SEGMENT DATA=my_data1
int i1;
#pragma SEGMENT PROG=my_prog2 DATA=my_data2
int i2;         /* This is in  the "active" static data segment. */

void function()
{
  i1 += i2;     /* i2 is in "active" static data segment. */
}

#pragma SEGMENT DATA=my_prog1
main()
{
  long a1;      /* a1 is a dynamic variable on the stack. */

  a1 = 1;
  i1++;
  i2++;
  function();
}
```

Figure 1-8. mediummodel.c

When **mediummodel.c** is compiled using the following command line, the **mediummodel.O** listing file results.

```
$  cc8086 -SOLm medium mediummodel.c <RETURN>
```

```
HPB1493-19303  8086 C CROSS COMPILER  A.04.01  mediummodel.c

;
;MKT:@(#) B1493-19303 A.04.01 8086 C CROSS COMPILER                    03May95

; Memory Model:  medium
;
$PAGEWIDTH(230)
$NOPAGING
        NAME    "mediummodel"
%DEFINE(MM_CHECK_)(MM_CHECK_M)
%DEFINE(lib)(lib)
%DEFINE(SS)(DS)
%DEFINE(DS)(data_const)
%DEFINE(GRP)(data_const)
%DEFINE(ALIGN)(WORD)
%DEFINE(DALIGN)(WORD)
%GRP    GROUP data,idata,udata,const
data    SEGMENT WORD PUBLIC
data    ENDS
idata   SEGMENT WORD PUBLIC
idata   ENDS
udata   SEGMENT WORD PUBLIC
udata   ENDS
const   SEGMENT WORD PUBLIC
const   ENDS
my_prog2        SEGMENT %ALIGN PUBLIC 'CODE'
%DEFINE(CodeSegment)(my_prog2)
    1   #pragma SEGMENT DATA=my_data1
    2   int i1;
    3   #pragma SEGMENT PROG=my_prog2 DATA=my_data2
    4   int i2;         /* This is in  the "active" static data segment. */
    5
    6   void function()
    7   {
        PUBLIC  _function
        ASSUME  CS:%CodeSegment,DS:%GRP
_function       PROC    FAR
%SET(SAVE_ALL_NPX,2)
    8     i1 += i2;     /* i2 is in "active" static data segment. */
        MOV     AX,%DS:WORD PTR _i2[0]
        ADD     %DS:WORD PTR _i1[0],AX
    9   }
functionExit1:
returnLabel1:
        RET
_function       ENDP
   10
   11   #pragma SEGMENT DATA=my_prog1
   12   main()
   13   {
        PUBLIC  _main
        ASSUME  CS:%CodeSegment,DS:%GRP
_main   PROC    FAR
%SET(SAVE_ALL_NPX,2)
        PUSH    BP
        MOV     BP,SP
```

Figure 1-9. mediummodel.O

```
        SUB     SP,4
%SET(S_a1,-4)
   14     long a1;      /* a1 is a dynamic variable on the stack. */
   15
   16     a1 = 1;
        MOV     SS:WORD PTR [BP+%S_a1+0],1
        MOV     SS:WORD PTR [BP+%S_a1+0+2],0
   17     i1++;
        INC     %DS:WORD PTR _i1[0]
   18     i2++;
        INC     %DS:WORD PTR _i2[0]
   19     function();
        CALL    FAR PTR _function
   20   }
functionExit2:
        MOV     SP,BP
        POP     BP
returnLabel2:
        RET
_main   ENDP
my_prog2        ENDS
my_data1        SEGMENT %DALIGN PUBLIC
        PUBLIC  _i1
        EVEN
_i1     LABEL   BYTE
        DB      2 DUP(0)
my_data1        ENDS
my_data2        SEGMENT %DALIGN PUBLIC
        PUBLIC  _i2
        EVEN
_i2     LABEL   BYTE
        DB      2 DUP(0)
my_data2        ENDS
        EXTRN   %MM_CHECK_:BYTE
mm_check        SEGMENT BYTE COMMON
        DW      OFFSET %MM_CHECK_
mm_check        ENDS
        END
```

Figure 1-9. mediummodel.O (continued

Calling Run-Time and Support Libraries

Run-time library routines are called *implicitly* by the generated assembly code. For example, with large and medium memory model, **ADD_F32A_LM, ADD_F32B_L**, or **ADD_F32C_L** would be called to add two floats. Which of the three routines the compiler actually uses depends on where the arguments are found and how the code is being optimized. For the small adn compact memory model **ADD_F32A_SC, ADD_F32B_S,** or **ADD_F32C_S** would be called. Note that the names are different between memory models to guarantee that the correct run-time library is used.

Since these implicitly called routines are not visible in the C source, a special segment named **lib** is reserved and understood by the compiler to be the segment in which the run-time library is defined. (**lib** is replaced with **prog** for the small and compact memory model).

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. For the large and medium memory model, their segment names are the same as the base name of the library (e.g., **libc.a**'s segment is **libc**). For the small and compact model, the segment name is **prog**, the same as with user-written code.

The libcalls.c listing shows the calling of run-time library routines and the calling of a support library routine. Note that it is important to use **# include < stdio.h>** since without it the compiler does not know that **printf()** is in named segment **libc**. When **libcalls.c** is compiled using the following command line, the **libcalls.O** listing file results.

```
$ cc8086 -SOL libcalls.c
```

```
#include <stdio.h>
main()
{
  float f = 1.0;
  float g = 1.0;

  printf("Sum is %f\n", f+g);
}
```

Figure 1-10. libcalls.c

```
HPB1493-19303  8086 C Cross Compiler A.04.01  libcalls.c

;
;MKT:@(#) B1493-19303 A.04.01 8086 C CROSS COMPILER                    03May95
; Memory Model:  large
;
$PAGEWIDTH(230)
$NOPAGING
        NAME    "libcalls"
%DEFINE(MM_CHECK_)(MM_CHECK_L)
%DEFINE(lib)(lib)
%DEFINE(SS)(SS)
%DEFINE(DS)(DS)
%DEFINE(ALIGN)(WORD)
%DEFINE(DALIGN)(WORD)
prog_libcalls   SEGMENT %ALIGN PUBLIC 'CODE'
%DEFINE(CodeSegment)(prog_libcalls)
    1   #include <stdio.h>
    2   main()
    3   {
        PUBLIC  _main
        ASSUME  CS:%CodeSegment,DS:NOTHING
_main   PROC    FAR
%SET(SAVE_ALL_NPX,2)
        PUSH    BP
        MOV     BP,SP
        SUB     SP,8
%SET(S_f,-8)
        MOV     SS:WORD PTR [BP+%S_f+0],00H
        MOV     SS:WORD PTR [BP+%S_f+0+2],03F80H
%SET(S_g,-4)
        MOV     SS:WORD PTR [BP+%S_g+0],00H
        MOV     SS:WORD PTR [BP+%S_g+0+2],03F80H
    4     float f = 1.0;
    5     float g = 1.0;
    6
    7     printf("Sum is %f\n", f+g);
        LES     DI,SS:DWORD PTR [BP+%S_f+0]
        PUSH    ES
        PUSH    DI
        LES     DI,SS:DWORD PTR [BP+%S_g+0]
        PUSH    ES
        PUSH    DI
%lib    SEGMENT WORD PUBLIC 'CODE'
        EXTRN   ADD_F32A_LM:FAR
%lib    ENDS
        CALL    FAR PTR ADD_F32A_LM
        POP     AX
        POP     DX
        SUB     SP,8
%lib    SEGMENT WORD PUBLIC 'CODE'
        EXTRN   F32_TO_F64_LM:FAR
%lib    ENDS
        CALL    FAR PTR F32_TO_F64_LM
        MOV     DX,SEG String1
        MOV     AX,OFFSET String1+0
```

Figure 1-11. libcalls.O

```
        PUSH    DX
        PUSH    AX
        CALL    FAR PTR _printf
        ADD     SP,12
    8   }
functionExit1:
        MOV     SP,BP
        POP     BP
returnLabel1:
        RET
_main   ENDP
prog_libcalls   ENDS
        EXTRN   _printf:FAR
const   SEGMENT %DALIGN PUBLIC
String1 LABEL   BYTE
        DB      'Sum is '
        DB      37
        DB      'f'
        DB      10
        DB      0
const   ENDS
        EXTRN   %MM_CHECK_:BYTE
mm_check        SEGMENT BYTE COMMON
        DW      OFFSET %MM_CHECK_
mm_check        ENDS
        END
```

Figure 1-11. libcalls.O (continued)

To specify the target microprocessor

- Use the appropriate compiler command:

    - cc8086 for the 8086
    - cc80186 for the 80186

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** 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:

```
cc8086 simple.c -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

The **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 **cc8086** 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/cc8086
AS=/usr/hp64000/bin/as86
LD=/usr/hp64000/bin/ld86
```

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.

Assembling a **.s** file produced by the **cc8086** compiler requires that two programs be executed in succession, the assembly macro preprocessor (**ap86**) and the assembler (**as86**). Because of this, the implicit suffix rule **.s.o:** cannot be used; you should explicitly put a **.s.o:** rule in your makefile:

```
.s.o:
          $(CC)  $(CFLAGS)  -c  $*.s
```

A second difference involves the implicit rule **.c:** which tells make to build to an executable file from a C source file. **Make** expects that the executable file has no suffix, but the 8086/186 cross language tools expect a **.x** suffix (**.X** if **-h** option has been selected). The solution is to simply specify your own suffix rule (**.c.x:** or **.c.X:**) which performs the functionality of the **.c:** implicit rule:

```
.c.X:
          $(CC)  $(CFLAGS)  -o $*.X  $*.c
```

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 cc8086 cross tools.

CC = cc8086

# 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:

```
cc8086 -O -c main.c
cc8086 -O -c file1.c
yacc grammar.y
cc8086 -O -c y.tab.c
rm y.tab.c
```

```
mv y.tab.o grammar.o
cc8086 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**, the startup routines **crt0.o** or **crt1.o**, or the monitor stub **mon_stub.o**:

1 Set up directories for the different memory models.

Select a directory from which you expect to run the compiler. The **make** utility will be used to create new environment-dependent libraries and object files which contain the changes made to the source files. As provided, **Makefile** assumes there are five directories named **src, large**, **medium**, **compact** and **small** in a parent directory. Makefile expects to be located in, and run from the **src** directory. The object and library-archive files will be built in the **large, medium, compact,** and **small** directories; the emulation monitor is built in the parent directory (the directory you are currently in). The following command sets up the needed directories.

```
$ mkdir src large medium compact small
```

2 Copy the source files.

The following command copies the environment-dependent source files to the current directory.

```
cp /usr/hp64000/env/hp<emul_env>/src/*  src
```

3 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.

```
cd  src
chmod  644  *
```

Now you may edit the source files as needed.

4 Run the "make" command.

The following command will create, for both large and small memory models, new environment-dependent library files, **env.a**, new startup and error-handling modules, **crt0.o**, **crt1.o**, **init_stub.o**, and **div_by_0.o**, and a new emulation monitor module, **monitor.o**, which is common to both memory models.

```
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
```

Now return to the parent directory.

```
cd  ..
```

5 Modify the default linker command file.

The following 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<emul_env>/large/iolinkcom.k  large
chmod  644  large/iolinkcom.k
vi  large/iolinkcom.k
```

Change *all* lines which read:

```
LOAD  /usr/hp64000/env/hp<emul_env>/large/env.a
```

to

```
LOAD  large/env.a
```

If small memory model is being used, do the same procedure as for the large memory except substitute **small** for **large** in all the commands.

If the medium or compact memory model is used, follow the same procedure as for the large memory model, except substitute "medium" or "compact" for "large" in all the commands.

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.

If no emulation monitor is needed, the LOAD command for **monitor.o** may be commented out or removed. The **env.a** library, which is loaded after **monitor.o**, will resolve the necessary external symbols. Note that **monitor.o**, when used, must be loaded before **env.a**.

Note The environment for HP 64700 series emulators for the Intel family does not include a **monitor.o** file. These emulators use a background monitor which does not need to be linked to the user's code.

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. Remember to use **-k** < *memory model*> **/iolinkcom.k** to get the appropriate modified linker command file.

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** and are designed to support the emulator (and simulator, if available). But these do more than just help you use the emulator.

The 8086/186 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 8086/186 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 pages

- Use the host operating system's **man** command.

You can display on-line "man pages" for any of the programs which make up the 8086/186 C Cross Compiler:

- **cc8086**
- **cpp8086**
- **clst8086**

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 *cc8086* man page and keep it near your computer.

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.

Example

To view the **cc8086** on-line manual page, just type in the following command from the operating system prompt:

```
man  cc8086
```

Information on the **cc8086** compiler syntax and options will be scrolled onto your display.

Part 2

Compiler Reference

Part 2

2

C Compilation Overview

An overview of the 8086/186 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 8086/186 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 8086/186 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 8086/186 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 **cc8086** cross compiler and not to the native host **cc** compiler.

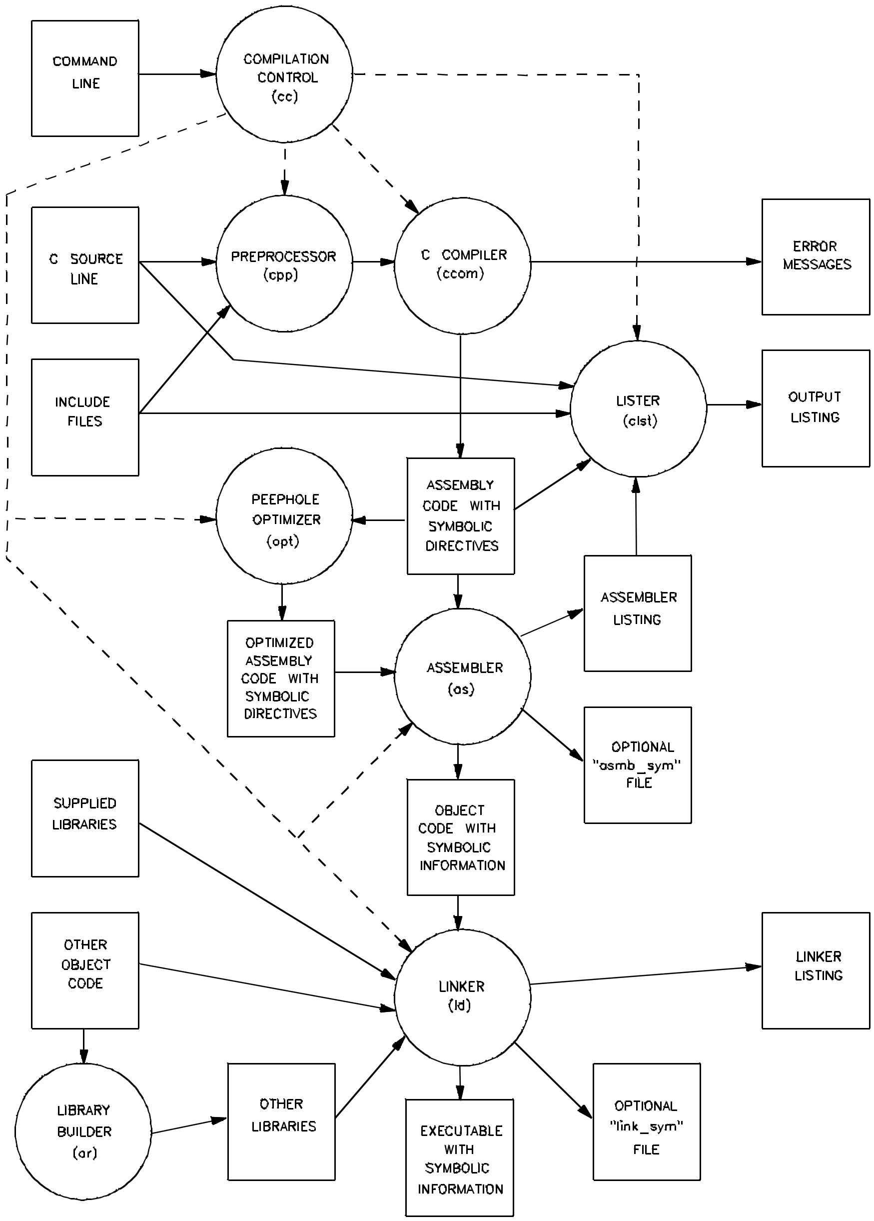

Figure 2-1. C Compilation Overview

Note

When you use the cc8086 command, the 8086/186 C compiler generates 8086 code. When you use the cc80186 command, the 8086/186 C compiler generates 80186 instructions where it is *optimal* to do so. In cases where 80186-specific instructions have no advantage over 8086 instructions, 8086 instructions are generated.

Except for the generation of 80186 instructions, the 8086 and 80186 compilers are identical.

Throughout the remainder of the manual, this product is referred to as the 8086/186 C compiler. You should take that to mean 8086/186 C Cross Compiler.

Compilation Control Routine

The entire system is controlled by a compilation control routine, cc8086 (or cc80186 for the 80186). The compilation control routine calls in sequence: the C preprocessor (cpp8086), the C compiler (ccom8086S/C/M/L), optionally the peephole optimizer (opt8086), the assembly macro preprocessor (ap86), the assembler (as86), optionally the lister (clst8086), and the linker (ld86). Many of these programs may be run individually using the cc8086 command's options. See the on-line man pages for the description of the command syntax and options.

The librarian (ar86) is a separate tool for building archive files used by the linker.

C Preprocessor

The 8086/186 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 8086/186 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 <u>not</u> a stand alone tool for use on hand-written assembly code.

Refer to the "Optimizations" chapter for more information on the peephole optimizer.

Assembly Preprocessor

The assembly preprocessor is the HP B1449 assembly preprocessor which accepts an assembly language source file (optionally containing symbolic debug information defined by special directives) and produces another assembly language file which has all assembly preprocessor macros, etc. expanded. The 8086/186 C compiler generates assembly preprocessor macros; therefore, assembly language code generated by the 8086/186 C compiler must pass through the assembly preprocessor before being assembled.

Assembler

The assembler is the HP B1449 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 clst8086 command syntax and options.

Librarian

The librarian is the HP B1449 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 B1449 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 B1493 8086/186 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 8086/186 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 8086/186 C compiler understands are listed below. Pragmas which are not recognized cause a warning message to be written to the standard error output.

pragma SEGMENT

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

pragma DS

Provides for re-specifying which segment will be accessed DS-relative. (Refer to the "Segment 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 segment (see the "# pragma SEGMENT" 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^{16}-2 bytes of local variables in one function block.
- Unlimited simultaneous macro definitions.
- Limited to 2^{16}–2 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.
- A single object may be as large as 2^{16}–2 bytes in size.
- Unlimited nesting levels of include files.
- Unlimited number of cases in a switch statement.
- Size of a switch statement body is limited to 2^{16}–1 bytes of generated code.

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	16	–32768 to 32767	0 to 65535
long	32	–2147483648 to 2147483647	0 to 4294967295
float	32	+ /– 1.18 x 10^{-38} to + /– 3.4 x 10^{38}	
double	64	+ /– 2.23 x 10^{-308} to + /– 1.8 x 10^{308}	

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 (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 1.18×10^{-38} to 3.4×10^{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 2.23×10^{-308} to 1.8×10^{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 8087" 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 "Small Memory Model Run-Time Routines" and "Large Memory Model Run-Time Routines" chapters) or to math library routines which eventually call run-time library routines. With the "generate code for the 8087" command line option, most real number operations are performed in-line with 8087 instructions.

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

Without the 8087. 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 8087. When the "generate code for the 8087" (**-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 8087 allows you to control its precision, rounding, trapping, and infinity behaviors. You may change the behavior of the 8087 by using the **_set_fp_control()** function, which is described under **_fp_error** in the "Libraries" chapter.

Characters

In addition to the **char** type, the 8086/186 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	16-bits (Small memory model); 32-bits (Large memory model).
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 or 2 bytes depending on the constant values of the elements.

Pointers

Pointers are addresses which point to stored values. Pointers occupy four bytes (two bytes for the small memory model) and are aligned on two byte boundaries. 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    */
                        /* at 2-byte aligned address.              */
main()
{
        int     array[4][7];    /* 4*7*2 = 56 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 segment (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 segment 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 segment (if the array's type is not "**const**") or in the named CONST linker segment (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 two bytes. For example, a structure whose members are a **char**, an **int**, and a **double** would be allocated 12 bytes (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 {        /* 12 bytes of storage allocated at 2-byte boundary. */
        char    c;      /* First byte of structure.  */
        int     i;      /* Begins at 3rd byte of structure.  */
        double  d;      /* Begins at 5th byte of structure.  */
} var;

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

When the "byte align data" option is used, there will be no alignment padding between members or at the end of a structure. The structure size may be other than a multiple of two bytes.

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** (word) boundary.

Examples of bit field declarations follow.

```
struct {
        int             f1:4;     /* f1 is a signed bit field,     */
                                  /* occupying bits 0-3 of the     */
                                  /* first word.                   */
                                  /*                               */
        unsigned        :8;       /* 8 bits of padding occupy      */
                                  /* bits 4-11 of the first        */
                                  /* word.                        */
                                  /*                              */
        unsigned        f2:8;     /* f2 occupies bits 12-15 of the */
                                  /* first word and bits           */
                                  /* 0-3 of the second word.       */
                                  /*                               */
        int             :0,f3:7;  /* f3 occupies bits 0-6 of       */
                                  /* the third word.               */
} a;                              /* The size of the structure is  */
                                  /* 6 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 {
        float           fp_rep;
        struct {
                unsigned int      lowbits;
                unsigned int      :15;
                unsigned int      sign : 1;
        } parts
} fp_num;

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

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, it is an error.

An **enum** typed variable can be used in expressions wherever integral typed variables are allowed. An enumerated constant is always of type integer. 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 two bytes means that a variable's absolute address is a multiple of 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 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 "byte align data" (**-Q**) option. In the presence of this option, data is aligned at byte boundaries.

The following table summarizes the default alignment and padding of the various data types when the "byte align data" option is not used.

Note If the "byte align data" option is used, alignment is always 1 and there is no padding.

Table 3-2. Arithmetic Data Type Alignment

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

Alignment Examples

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

Default alignment dictates that a **char** variable followed by an **int** variable "wastes" one byte 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 one byte is "wasted" when a **char** variable is followed by a structure.

The **sizeof** bytestruct declared with:

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

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

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

is four (one for element1, one "wasted" for alignment, two for element2, and none for padding as the size is a multiple of two).

4

Compiler Generated Assembly Code

Description of the assembly code generated by the compiler.

The compiler generates assembly code for the HP B1449 assembly macro preprocessor (ap86) and assembler (as86). 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
- Memory model mismatch 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 cc8086 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 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 B1449 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.

- Space is reserved for a structure result (if the size returned is greater than 4 bytes).
- Parameters are pushed (last is pushed first).
- A pointer to the result address is pushed (if size returned is greater than 4 bytes).
- The subroutine call is made and the return address is pushed.
- The old frame pointer is pushed.

- Space for automatics (locals) is allocated.
- The old Data Segment (register DS) is pushed and DS is loaded with the new data segment paragraph number. (Large memory model only.)
- The old register variable (register SI) is pushed to buffer its value.
- The complete internal state of the 8087 is pushed *if* the "generate code for the 8087" command line option was used and one or more floating-point register variables are used in the function.
- 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, the 8087 state, if it was saved, is restored; any 8087 registers which were saved are restored; 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.

High Address	Used stack space	
	Reserved space for structure result	Absent if result is < = 4 bytes or if result is returned through a variable.
	Last parameter ⇑ First parameter	Absent if no parameters are passed. (Last passed parameter is pushed first.)
	[segment] Result [offset] address	Absent if size returned is < = 4 bytes. (Address size is 2 words.)
	[segment] Return [offset] address	(Address size is 2 words.)
Frame pointer (BP)	Old frame pointer (BP)	Absent if there are no parameters or locals. (Size is 1 word.)
	Last local ⇑ First local	Absent if function does not declare any local (automatic) variables. (Last declared local is first on stack.)
	Buffered data segment (DS)	Absent if function does not access DS-relative static data.
	Buffered register variable (SI)	Absent if function does not use register variables.
	8087 register variables	Present when "-f" option is used *and* 8087 register variables are used
Stack pointer (SP)	Temporaries ⇓	Stack changes as temporaries are saved and used in expressions.
Low Address	Top of stack	

Figure 4-2. Stack Frame Format

```
HPB1493-19303  8086 C Cross Compiler A.04.01  esfm.c

;
;MKT:@(#) B1493-19303 A.04.01 8086 C CROSS COMPILER                    03May95

; Memory Model:  large
;
$PAGEWIDTH(230)
$NOPAGING
        NAME    "esfm"
%DEFINE(MM_CHECK_)(MM_CHECK_L)
%DEFINE(lib)(lib)
%DEFINE(SS)(SS)
%DEFINE(DS)(DS)
%DEFINE(ALIGN)(WORD)
%DEFINE(DALIGN)(WORD)
prog_esfm       SEGMENT %ALIGN PUBLIC 'CODE'
%DEFINE(CodeSegment)(prog_esfm)
    1   typedef struct {
                int     month,day,year;
    3   } date;
    4
    5   int year = 87;
    6
    7   main()
    8   {
        PUBLIC  _main
        ASSUME  CS:%CodeSegment,DS:data
_main   PROC    FAR
%SET(SAVE_ALL_NPX,2)
        PUSH    BP
        MOV     BP,SP
        SUB     SP,6
        PUSH    DS
        MOV     AX,data
        MOV     DS,AX
%SET(S_d,-6)
    9           date    d,set_date();
   10
   11           set_date(d,5,18,year);
        SUB     SP,6
        PUSH    %DS:WORD PTR _year[0]
        MOV     AX,18
        PUSH    AX
        MOV     AX,5
        PUSH    AX
        PUSH    SS:WORD PTR [BP+%S_d+0+4]
        PUSH    SS:WORD PTR [BP+%S_d+0+2]
        PUSH    SS:WORD PTR [BP+%S_d+0]
        MOV     AX,SP
        ADD     AX,12
        PUSH    SS
        PUSH    AX
        CALL    FAR PTR _set_date
        ADD     SP,6+0+12+4
   12   }
functionExit1:
```

Space reserved for structure result.

Parameters pushed.

Structure result address pushed.

Function call.

Stack pointer incremented (parameters popped).

Figure 4-3. Example Stack Frame Management Code

```
        POP     DS
        MOV     SP,BP
        POP     BP
returnLabel1:
        RET
_main   ENDP
   13
   14   date set_date(x,mo,da,yr)
   15   date    x;
   16   int     mo,da,yr;
   17   {
        PUBLIC  _set_date
        ASSUME  CS:%CodeSegment,DS:NOTHING
_set_date       PROC    FAR
%SET(SAVE_ALL_NPX,2)
        PUSH    BP
        MOV     BP,SP
        SUB     SP,16
        PUSH    SI
%SET(S_x,10)
%SET(S_mo,16)
%SET(S_da,18)
%SET(S_yr,20)
%SET(S_i1,-16)
%SET(S_i2,-8)
;S_i3 is in register SI.
   18           double          i1,i2;
   19           register int    i3;
   20
   21           x.month = mo;
        MOV     AX,SS:WORD PTR [BP+%S_mo+0]
        MOV     SS:WORD PTR [BP+%S_x+0],AX
   22           x.day = da;
        MOV     AX,SS:WORD PTR [BP+%S_da+0]
        MOV     SS:WORD PTR [BP+%S_x+2],AX
   23           x.year = yr;
        MOV     AX,SS:WORD PTR [BP+%S_yr+0]
        MOV     SS:WORD PTR [BP+%S_x+4],AX
   24           return(x);
        PUSH    SS:WORD PTR [BP+%S_x+0+4]
        PUSH    SS:WORD PTR [BP+%S_x+0+2]
        PUSH    SS:WORD PTR [BP+%S_x+0]
        LES     DI,SS:DWORD PTR [BP+6]
        CLD
        POP     AX
        STOSW
        POP     AX
        STOSW
        POP     AX
        STOSW
   25   }
functionExit2:
        POP     SI
        MOV     SP,BP
        POP     BP
returnLabel2:
        RET
_set_date       ENDP
```

Old frame pointer pushed and space for locals allocated.

Structure result returned.

Function exit.

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

```
prog_esfm        ENDS
data    SEGMENT %DALIGN PUBLIC
        PUBLIC  _year
        EVEN
_year   LABEL   BYTE
        DW      87
data    ENDS
        EXTRN   %MM_CHECK_:BYTE
mm_check        SEGMENT BYTE COMMON
        DW      OFFSET %MM_CHECK_
mm_check        ENDS
        END
```

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

Structure Results

C allows functions to return results of type **struct**. Although most function results are returned in a working register (AL, AX, DL-AX, or DX-AX), structures greater in size than 4 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** 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

Within a called function's prolog, a PUSH BP instruction followed by a MOV BP,SP instruction (or just a ENTER instruction) is used to save the old frame pointer (BP) and set up the new stack frame. This occurs only if one or more of the following conditions is true:

- The "optimize" option is off.
- The "run-time error checking" option is on.
- Automatic variables exist for the current function.
- Parameters exist for the current function.
- The current function returns a value which has a size greater than 4 bytes. (This causes a "result address" to be placed on the stack.)

Reserving Space For "C" Variables

After the instructions for setting up the stack frame, any automatic variables and any register variables that cannot be assigned to the SI register are allocated by decrementing the stack pointer (SP). No stack pointer adjustment instruction will be generated if there are no automatic variables or unassigned register variables. (Total local space is padded to a multiple of two bytes.)

Pushing Data Segment (DS) Register

(Large and compact memory models only). Following the allocation of automatics, if the memory model is "large" and the current function references any static data, the data segment (DS) register will be pushed on the stack and then loaded with a new segment paragraph number. This is to allow the DS-relative accesses within the current function to address the appropriate static data segment. This code for setting up a new DS-relative static data segment will never appear for the small and medium memory models, and does not appear for the large memory model when there is no static data associated with the current function.

Buffering Register Variable (SI)

Next, the function prolog pushes the old register variable (SI) on the stack if SI has been allocated for use by the function as a register variable. Also, the compiler may use this register for an automatic regardless of whether or not it has been declared with the **register** storage class specifier (see the "Register Usage" section which follows).

Buffering 8087 Floating Point Register Variables

The last code in the function prolog saves floating point register variables ST(2) through ST(6). In general, it is more cost effective when saving several 8087 registers to save the whole 8087 state instead of saving individual registers. Therefore, if the 8087 is being used (that is, if the "generate code for the 8087" option is on) and one or more floating point register variables are to be used by the function then the complete 8087 internal state is saved into a 94-byte space on the stack. This is accomplished with an FSAVE instruction followed by an FLDCW instruction. The FLDCW instruction is necessary to propagate the previously set up 8087 control word into the reset 8087. (The FSAVE instruction also resets the 8087.) At function epilog the internal state of the 8087 is restored with instructions FSTCW and FRSTOR. Here, the FSTCW is required to propagate back any changes made to the 8087 control word while in this function.

Observe that the 8087 status word is not propagated when 8087 register variables are saved. This is normally not a problem, except when exceptions are masked that are to be later unmasked and acted upon. These pending exceptions might not be retained outside of the function where they occur. This loss of "exceptions history" occurs only when the "generate code for the 8087" option is on and floating point register variables are used.

For interrupt routines, if the "generate code for the 8087" option is on the complete 8087 internal state is saved regardless of whether or not floating point register variables are used. Also, the 8087 control word is not propagated, so the interrupt routine writer must set up the control word before using the 8087.

Accessing Parameters

Each parameter's assembly symbol name is SET to that parameter's offset from the frame pointer. The value of these offsets differ from one stack model to another. Refer to the "Stack Models" chapter for illustrations of the stack models.

In the stack shown in figure 4-2, the offset of the first parameter will be 6 (large memory model) if the value returned is 4 bytes or less. The offset of the first parameter will be 10 (large memory model) if the result size is greater than 4 bytes. For example, if "p" is the first parameter passed, the compiler may generate the following line in the assembly:

```
%SET(S_p,6)
```

Parameters are accessed by using the symbol names relative to BP. Notice that when referencing a parameter, a percent sign (%) must precede the parameter name. For example:

```
MOV SI,SS:WORD PTR [BP+%S_p+0]
```

Shortening Parameters

Unless function prototypes are used (see the "ANSI Extensions to C" section in the "C Compiler Overview" chapter), parameters of type **char** are widened to **int** when passed. Thus, any parameters formally declared to be of type **char** must be shortened from **int.** Since this shortening is defined to be by truncation, it is accomplished by simply using the parameter as if it were a **char**. (The parameter’s offset needs no adjusting.)

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.

```
HPB1493-19303  8086 C Cross Compiler A.04.01  parmshrt.c

;
;MKT:@(#) B1493-19303 A.04.01 8086 C CROSS COMPILER                    03May95

; Memory Model:  large
;
$PAGEWIDTH(230)
$NOPAGING
        NAME    "parmshrt"
%DEFINE(MM_CHECK_)(MM_CHECK_L)
%DEFINE(lib)(lib)
%DEFINE(SS)(SS)
%DEFINE(DS)(DS)
%DEFINE(ALIGN)(WORD)
%DEFINE(DALIGN)(WORD)
prog_parmshrt   SEGMENT %ALIGN PUBLIC 'CODE'
%DEFINE(CodeSegment)(prog_parmshrt)
    1   main()
    2   {
        PUBLIC  _main
        ASSUME  CS:%CodeSegment,DS:NOTHING
_main   PROC    FAR
%SET(SAVE_ALL_NPX,2)
        PUSH    BP
        MOV     BP,SP
        SUB     SP,6
%SET(S_c,-6)
%SET(S_f,-4)
    3           char    c, char_funct();
    4           float   f, float_funct();
    5
    6           char_funct(c);
        MOV     AL,SS:BYTE PTR [BP+%S_c+0]
        CBW
        PUSH    AX
        CALL    FAR PTR _char_funct
        POP     CX
    7           float_funct(f);
        LES     DI,SS:DWORD PTR [BP+%S_f+0]
        MOV     DX,ES
        XCHG    AX,DI
        SUB     SP,8
%lib    SEGMENT WORD PUBLIC 'CODE'
        EXTRN   F32_TO_F64_LM:FAR
%lib    ENDS
        CALL    FAR PTR F32_TO_F64_LM
        CALL    FAR PTR _float_funct
        ADD     SP,8
    8   }
functionExit1:
        MOV     SP,BP
        POP     BP
returnLabel1:
        RET
_main   ENDP
```

char widened to **int**.

float widened to **double**

Figure 4-4. Widening and Shortening of Parameters

```
     9
    10   char char_funct(chr)
    11   char    chr;
    12   {
         PUBLIC  _char_funct
         ASSUME  CS:%CodeSegment,DS:NOTHING
 _char_funct     PROC    FAR
 %SET(SAVE_ALL_NPX,2)
         PUSH    BP
         MOV     BP,SP
 %SET(S_chr,6)
    13           chr = 'A';
         MOV     SS:BYTE PTR [BP+%S_chr+0],65
    14           return(chr);
         MOV     AL,65
    15   }
 functionExit2:
         POP     BP
 returnLabel2:
         RET
 _char_funct     ENDP
    16
    17   float float_funct(flt)
    18   float   flt;
    19   {
         PUBLIC  _float_funct
         ASSUME  CS:%CodeSegment,DS:NOTHING
 _float_funct    PROC    FAR
 %SET(SAVE_ALL_NPX,2)
         PUSH    BP
         MOV     BP,SP
         SUB     SP,4
 %SET(S_flt,-4)
 %SET(S_wide_param1,6)
         PUSH    SS:WORD PTR [BP+%S_wide_param1+0+6]
         PUSH    SS:WORD PTR [BP+%S_wide_param1+0+4]
         PUSH    SS:WORD PTR [BP+%S_wide_param1+0+2]
         PUSH    SS:WORD PTR [BP+%S_wide_param1+0]
 %lib    SEGMENT WORD PUBLIC 'CODE'
         EXTRN   F64_TO_F32_LM:FAR
 %lib    ENDS
         CALL    FAR PTR F64_TO_F32_LM
         MOV     SS:WORD PTR [BP+%S_flt+0],AX
         MOV     SS:WORD PTR [BP+%S_flt+0+2],DX
    20           flt = 1.0;
         MOV     SS:WORD PTR [BP+%S_flt+0],00H
         MOV     SS:WORD PTR [BP+%S_flt+0+2],03F80H
    21           return(flt);
         LES     DI,SS:DWORD PTR [BP+%S_flt+0]
         MOV     DX,ES
         XCHG    AX,DI
    22   }
 functionExit3:
         MOV     SP,BP
         POP     BP
 returnLabel3:
         RET
 _float_funct    ENDP
 prog_parmshrt   ENDS
```

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

double shortened to **float.**

```
HPB1493-19303  8086 C Cross Compiler A.04.01  protypes.c

;
;MKT:@(#) B1493-19303 A.04.01 8086 C CROSS COMPILER                    03May95

; Memory Model:  large
;
$PAGEWIDTH(230)
$NOPAGING
        NAME    "protypes"
%DEFINE(MM_CHECK_)(MM_CHECK_L)
%DEFINE(lib)(lib)
%DEFINE(SS)(SS)
%DEFINE(DS)(DS)
%DEFINE(ALIGN)(WORD)
%DEFINE(DALIGN)(WORD)
prog_protypes   SEGMENT %ALIGN PUBLIC 'CODE'
%DEFINE(CodeSegment)(prog_protypes)
    1   main()
    2   {
        PUBLIC  _main
        ASSUME  CS:%CodeSegment,DS:NOTHING
_main   PROC    FAR
%SET(SAVE_ALL_NPX,2)
        PUSH    BP
        MOV     BP,SP
        SUB     SP,6
%SET(S_c,-6)
%SET(S_f,-4)
    3           char    c, char_funct(char);
    4           float   f, float_funct(float);
    5
    6           char_funct(c);
        MOV     AL,SS:BYTE PTR [BP+%S_c+0]
        PUSH    AX
        CALL    FAR PTR _char_funct
        POP     CX
    7           float_funct(f);
        LES     DI,SS:DWORD PTR [BP+%S_f+0]
        PUSH    ES
        PUSH    DI
        CALL    FAR PTR _float_funct
        ADD     SP,4
    8   }
functionExit1:
        MOV     SP,BP
        POP     BP
returnLabel1:
        RET
_main   ENDP
    9
   10   char char_funct(
   11   char    chr)
```

char no longer widened to **int**.

float no longer widened to **double.**

Figure 4-5. Function Prototype Parameter Passing

```
   12   {
        PUBLIC  _char_funct
        ASSUME  CS:%CodeSegment,DS:NOTHING
_char_funct     PROC    FAR
%SET(SAVE_ALL_NPX,2)
        PUSH    BP
        MOV     BP,SP
%SET(S_chr,6)
   13           chr = 'A';
        MOV     SS:BYTE PTR [BP+%S_chr+0],65
   14           return(chr);
        MOV     AL,65
   15   }
functionExit2:
        POP     BP
returnLabel2:
        RET
_char_funct     ENDP
   16
   17   float float_funct(
   18   float   flt)
   19   {
        PUBLIC  _float_funct
        ASSUME  CS:%CodeSegment,DS:NOTHING
_float_funct    PROC    FAR
%SET(SAVE_ALL_NPX,2)
        PUSH    BP
        MOV     BP,SP
%SET(S_flt,6)
   20           flt = 1.0;
        MOV     SS:WORD PTR [BP+%S_flt+0],00H
        MOV     SS:WORD PTR [BP+%S_flt+0+2],03F80H
   21           return(flt);
        LES     DI,SS:DWORD PTR [BP+%S_flt+0]
        MOV     DX,ES
        XCHG    AX,DI
   22   }
functionExit3:
        POP     BP
returnLabel3:
        RET
_float_funct    ENDP
prog_protypes   ENDS
        EXTRN   %MM_CHECK_:BYTE
mm_check        SEGMENT BYTE COMMON
        DW      OFFSET %MM_CHECK_
mm_check        ENDS
        END
```

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

Accessing Locals

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

```
%SET(S_r,-20)
```

Local variables are accessed using the symbol name relative to BP. Notice that when referencing a local (automatic) variable, a percent sign (%) must precede the variable name. For example:

```
MOV SS:WORD PTR[BP+%S_r+0],DX
```

Using the Stack for Temporary Storage

Code generated by the function's 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

Function return values of one, two, three, or four bytes are returned in working registers AL, AX, DL-AX, or DX-AX respectively. Results greater in size 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.

Function Exit

At function exit, if the 8087 state was saved, it is restored. If the register variable (SI) has been buffered it is popped. If the data segment register (DS) has been buffered and altered, it is popped. And finally, if there is a stack frame, it is removed by adjusting the stack pointer past the automatics (if any) and popping the old frame pointer back into BP. 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. Function exit behavior may be modified by using the pragmas described in this chapter.

Register Usage

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.)

For the small memory and medium models, registers AX, BX, CX, DX, and DI are reserved as working registers for use in holding intermediate values of calculations. For the large and compact memory models, the working registers include those registers used for the small memory model and additionally register ES. Function return values of one, two, three, or four bytes are returned in working registers AL, AX, DL-AX, or DX-AX respectively. Larger types are returned on the stack. Registers BP and SP are the frame pointer and stack pointers.

For all memory models, the compiler will use the lower byte of registers AX, BX, CX, and DX (registers AL, BL, CL, and DL) to hold **char**s. The compiler also pairs up word registers in multiple combinations to create pseudo 32-bit registers for holding **longs, floats,** and additionally, pointers when large or compact memory model is in effect. For the small and medium memory model, these pseudo registers are in MSW-LSW order: DX-AX, AX-BX, CX-BX, DX-BX, AX-DI, CX-DI, and DX-DI. If the "byte align data" compiler option is used, the register pair DL-AX is used for 24-bit data. The large and compact memory models include those pseudo registers used by the small memory model as well as ES-BX and ES-DI. All four memory models also include a 64-bit pseudo register CX-BX-DX-AX.

Register Variable SI

Register SI is allocated by the compiler for use as a register variable. For the small and medium memory model this register variable may be either an integer or a pointer; for the large and compact memory model, it can only be an integer because a pointer will not fit.

Using the priorities listed below, the compiler allocates one of the following types of objects to register SI:

1 The first variable (parameter or local) declared with **register** storage class.

2 A local non-static or function variable, or the address of a static variable, according to frequency of occurrence of the variable's name in the function.

Specific use of the **auto** storage class prevents a local variable from becoming a register variable.

To better understand the allocation scheme, consider the following example. Suppose 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 because it has "qualified" for consideration and has a greater number of occurrences.

Passing Data

For C functions, no registers are used to explicitly pass data **to** a called function. Data is passed implicitly by using segment registers (according to the memory model) to maintain segment bases across a call boundary. The following registers are used to explicitly pass data back to the caller: AL for 8-bit data, AX for 16-bit data, DL-AX for 24-bit data, and the DX-AX pair for 32-bit data. All other return data is passed back via the stack. No other registers are used explicitly for passing data back to the caller.

The following registers must **not** be corrupted by a called routine (that is, the called routine must return with the same value sent by the caller):

- CS, DS, and SS segment registers for all four models.
- ES segment register for the small and medium memory model.
- SP, BP, and SI registers for all four memory models.

The compiler makes the following assumptions about segment registers, which affect whether a register will be reloaded or assumed to contain the needed value:

- CS does not change for the small and compact memory model.
- DS, SS, and ES contain the same value and do not change for the small and medium memory model.
- SS does not change for all four models.

8087 Registers

When using the "generate code for the 8087" option the compiler will use 8087 registers ST(0), ST(1), and ST(7) as working registers. The remaining five 8087 registers ST(2) through ST(6) are reserved for **float** and **double** register variables.

At code startup (crt0 or crt1) the 8087 is reset and its control word initialized. The 8087 NPX stack pointer (STP) is initialized to 0. STP can be from 0 to 7 and determines which 8087 hardware register is actually at "top of stack". Normally the compiler operates with STP equal to 0. When an object is to be loaded into the 8087 the compiler may "push" the object onto the 8087 "top of stack", causing STP to become 7. The compiler will eventually "pop" this value, with STP returning to 0. This "pushing" and "popping" effectively moves objects through register ST(7).

Registers ST(0) and ST(1) are general purpose working registers and are allowed to be either "empty" or contain a number, NaN, etc. Register ST(7) is a special purpose working register. The compiler expects that ST(7) will be "empty" except when the compiler is moving data through it. It is imperative that ST(7) be "empty" following in-line assembly code, or an 8087 "illegal operation" exception may occur. It is also required that in-line assembly code end with the 8087 NPX stack pointer (STP) in its original state (normally 0). In-line assembly code may leave registers ST(0) and ST(1) in any state.

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.

Memory Model Mismatch Checking

Because the compiler supports four different memory models it is important to distinguish code generated using one memory model from that generated with the other. Program modules compiled with small memory model may not be linked with modules compiled with large memory model. An attempt to do so will result in a link-time "unresolved symbol" error with the "memory model check" symbol.

Run-time library *lib* routines have different names from one memory model to the other. Small memory model routines end with "_S", compact memory model routines end with "_C", medium memory model routines end with "_M", and large memory model routines end with "_L". This guarantees that the wrong run-time library can never be accidentally linked to the user’s code. Many routines can be used by two memory models; thus "_LM" routines can be used by the large or medium memory model.

Code from compiled libraries, such as *libc* and *libm*, and the user's C code is guaranteed to be linkable only with modules compiled with the same memory model. This memory model checking is accomplished with a "memory model check" symbol which is different for each memory model. They symbol is **MM_CHECK_S** for the small memory model, **MM_CHECK_C** for compact, **MM_CHECK _M** for medium, and **MM_CHECK_L** for large. The memory model checking symbol adds only two bytes to the length of a program because the data word that holds the symbol is placed in a COMMON segment.

Figure 4-6 shows the assembly code which makes an external reference to the "memory model check" symbol. This symbol is defined by the startup code (crt0 or crt1) in the environment library (*env*). Thus, crt0 (or crt1) determines which memory model is expected to be in effect.

```
HPB1493-19303  8086 C Cross Compiler A.04.01  mmcheck.c

;
;MKT:@(#) B1493-19303 A.04.01 8086 C CROSS COMPILER                    03May95

; Memory Model:  large
;
$PAGEWIDTH(230)
$NOPAGING
        NAME    "mmcheck"
%DEFINE(MM_CHECK_)(MM_CHECK_L)
%DEFINE(lib)(lib)
%DEFINE(SS)(SS)
%DEFINE(DS)(DS)
%DEFINE(ALIGN)(WORD)
%DEFINE(DALIGN)(WORD)
prog_mmcheck    SEGMENT %ALIGN PUBLIC 'CODE'
%DEFINE(CodeSegment)(prog_mmcheck)
    1   main()
    2   {
        PUBLIC  _main
        ASSUME  CS:%CodeSegment,DS:NOTHING
_main   PROC    FAR
%SET(SAVE_ALL_NPX,2)
    3   }
functionExit1:
returnLabel1:
        RET
_main   ENDP
prog_mmcheck    ENDS
        EXTRN   %MM_CHECK_:BYTE
mm_check        SEGMENT BYTE COMMON
        DW      OFFSET %MM_CHECK_
mm_check        ENDS
        END
```

Figure 4-6. Memory Model Checking

Using Assembly Language in the C Source File

The 8086/186 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 for optimization purposes are forgotten. The register variable (SI), the frame pointer (BP), and the stack pointer (SP) are not buffered prior to embedded assembly language sections. You should buffer these registers if they will be used by 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 B1449 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

Figures 4-7, 4-8, and 4-9 give examples of using the **# pragma ASM/END_ASM** to embed assembly code in a C source file.

```
main()
{
  printf("Starting interrupt test.\n");
#pragma ASM
     INT 33     ;Interrupt handler is at 00084H.
#pragma END_ASM

  printf("Ending interrupt test.\n");
}


#pragma ASM
interrupt_table SEGMENT AT 8   ;Locate segment at 00080H.
     ORG 4                     ;Org to 00084H.
     DD _interrupt_handler
interrupt_table ENDS
#pragma END_ASM


#pragma INTERRUPT
static void interrupt_handler()
{
  printf("An interrupt 33 has occurred.\n");
}
```

Figure 4-7. # pragma ASM/END_ASM Example 1

```
/* Example of embedded assembly language code when using large memory model. */
main()
{
        auto int i1, i2;

        i1 = 1;
        i2 = get_global();

/* Swap i1 and i2 but do it in assembly. */
#pragma ASM
        MOV     AX,[BP+%S_i1] ;Percent needed for auto or parameter.
        XCHG    AX,[BP+%S_i2]
        MOV     [BP+%S_i1],AX
#pragma END_ASM

        printf("i1 = %d\ni2 = %d\n", i1, i2);
}


#pragma SEGMENT DATA=my_data
int global_var = 1234;
#pragma SEGMENT UNDO

int get_global()
                   {
                           register int reg_var;   /* reg_var is held in register SI. */

#pragma ASM
        PUSH    DS    ;Save current data segment.
        MOV     AX,SEG _global_var
        MOV     DS,AX
        MOV     SI,DS:WORD PTR _global_var ;Put it in reg_var.
        POP     DS     ;Restore data segment
#pragma END_ASM

        return(reg_var);
}
```

Figure 4-8. # pragma ASM/END_ASM Example 2

```
int *p;
int i;

main()
{
   p = &i;      /* Get address of i. */
   i++;         /* Increment i. */
   printf("Using C:  p = %p, i = %d\n",p,i);

   /* The following lines of assembly do the same thing as the lines   */
   /* "p = &i;" and "i++;" in C.  It illustrates the GROUP override    */
   /* requirements when embedding in-line assembly for small memory    */
   /* model.  The compiler defines both %DS and %GRP to be the group   */
   /* name "data_const" when using small memory model.  The macros     */
   /* could be replaced with "data_const" directly in this source but  */
   /* this could mean incompatibility with future releases of the      */
   /* compiler.  For large memory model %DS is defined to be just      */
   /* "DS"; %GRP is not defined.  Because %DS is available for both    */
   /* memory models it can be used to write assembly code that will    */
   /* work for both small and large memory models.  Note the "INC..."  */
   /* line of assembly.                                                */

   #pragma ASM
     #ifdef __SMALL_MODEL/* SMALL memory model */
        MOV  %DS:WORD PTR _p,OFFSET %GRP:_i
     #else/* LARGE memory model */
       ;Compiler has set up DS register to access p and i DS-relative.
       MOV  DS:WORD PTR _p,OFFSET _i
       MOV  DS:WORD PTR _p[2],SEG _i
     #endif
     INC  %DS:WORD PTR _i;For both small and large model.
   #pragma END_ASM
                    printf("Using assembly:  p = %p, i = %d\n",p,i);
                  }
```

Figure 4-9. # pragma ASM/END_ASM Example 3

__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 B1449 Assembler. The C compiler does not check the C_string for correctness. The compiler simply outputs the string to the assembly code.

Example

Figure 4-10 gives an example of using the **__asm** function.

```
/* Example of embedded assembly code when using large memory model. */

#define SAVE_DS __asm("\tPUSH   DS      ;Save current data segment.");
#define RESTORE_DS __asm("\tPOP DS      ;Restore data segment");

main()
{
  auto int i1, i2;

  i1 = 1;
  i2 = get_global();

/* Swap i1 and i2 but do it in assembly. */

/* Notice the "\t" white space 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.  Also, more than one assembly line may be handled     */
/* by a single __asm() function by separating the lines with a "\n".*/

  __asm("\tMOV  AX,[BP+%S_i1]  ;Percent needed for auto or parameter.");
  __asm("\tXCHG AX,[BP+%S_i2]\n\tMOV    [BP+%S_i1],AX");

  printf("i1 = %d\ni2 = %d\n", i1, i2);
}

#pragma SEGMENT DATA=my_data
int global_var = 1234;
#pragma SEGMENT UNDO

int get_global()
{
  register int reg_var;         /* reg_var is held in register SI. */

/* Notice the use of cpp macros to specify assembly code. */
  SAVE_DS
  __asm("\tMOV  AX,SEG _global_var");
  __asm("\tMOV  DS,AX");
  __asm("\tMOV  SI,DS:WORD PTR _global_var      ;Put it in reg_var.");
  RESTORE_DS

  return(reg_var);
}
```

Figure 4-10. __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 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 B1449 assembler. The C compiler does not check the C_string for correctness. The compiler simply outputs the string to the assembly code.

Example

Figure 4-11 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 IRET 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 IRET. But because the **# pragma FUNCTION_EXIT** code causes the routine to trap back to the operating system, it will never execute the IRET.

```
#pragma INTERRUPT
#pragma FUNCTION_EXIT "\tINT 2  ;Trap back to operating system."

static void interrupt_handler()
{
  printf("An interrupt 33 has occurred.\n");
}
/* Interrupt routine exits via "INT 2" instead of "IRET". */
```

Figure 4-11. # 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 **`cc8086 -SL`** and looking at the **.O** file. If you need to understand how the C preprocessor affected the code, use **`cc8086 -E`**.

Assembly Language and the Small Memory Model

When writing embedded assembly code in a C source file that is expected to be compiled using small memory model certain considerations must be made. For small memory model the compiler places all segments containing data or constants (data, idata, udata, heap, userstack, and const) into an assembly language group called **data_const.** The compiler then accesses objects in the **data_const** group "group-relative" instead of "segment-relative". For large memory model the compiler does "segment-relative" accesses to all data and constant objects because no segments are in a group. Objects contained in program segments (functions, for example) are always accessed (or called) "segment-relative", regardless of the memory model.

Figure 4-7 can be compiled using either memory model. It does not contain memory model dependent pragmas or assembly code.

Figure 4-8 can be compiled only for large memory model because it does contain a **SEGMENT** pragma and also contains memory model dependent assembly code. If compiled with small memory model the **SEGMENT** pragma would simply be warned at and ignored when encountered. However, the assembly instructions accessing *_global_var* would not produce functional code. Specifically, 1) the line "MOV AX,SEG _global_var" would load the AX register with the segment paragraph number of *_global_var* instead of the group paragraph number as it should, and 2) the line "MOV SI,DS:WORD PTR _global_var" may access *_global_var* as if it were not contained in a group and therefore go to the wrong place in memory.

Figure 4-9 demonstrates how to write assembly code that functions correctly no matter which memory model is used. At the beginning of the assembly file it produces, the compiler defines an assembly language macro *DS* to be either **DS** for large memory model (nothing needs to be changed), or **data_const** for small memory model. Thereafter, *%DS:* can be used instead of *DS:* to specify a "segment-relative" override for large memory model and at the same time a "group-relative" override for small memory model. For small memory model only, the compiler also defines another macro *GRP* to be **data_const**. This second macro allows embedded, "small memory model only" assembly code to reference the group name independently of the group name created by the compiler. Figure 4-9 shows its use as well.

5

Optimizations

Description of optimizations performed by the compiler.

The 8086/186 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 8086/186 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;) { statements }	⇒	i = 0 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 8086/186 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.

OPTIMIZED FOR SPACE (Default).

```
1   struct test {
2           int     a,b,c,d,e,f;
3   } x, y;
4
5   main()
6   {
  0000                                      PUBLIC  _main
  0000                                      ASSUME  CS:prog_space,DS:data
  0000                               _main  PROC    FAR
  0000
  0000   1E                                 PUSH    DS
  0001   B8 00 00           R               MOV     AX,data
  0004   8E D8                              MOV     DS,AX
7           y = x;
  0006   BA 00 00           R               MOV     DX,SEG _x
  0009   B8 00 00           R               MOV     AX,OFFSET _x+0
  000C   83 EC 0C                           SUB     SP,12
  000F   96                                 XCHG    AX,SI
  0010   8C DB                              MOV     BX,DS
  0012   8E DA                              MOV     DS,DX
  0014   8B FC                              MOV     DI,SP
  0016   8C D1                              MOV     CX,SS
  0018   8E C1                              MOV     ES,CX
  001A   B9 06 00                           MOV     CX,6
  001D   FC                                 CLD
  001E   F3 A5                              REP MOVSW
  0020   8E DB                              MOV     DS,BX
  0022   8B F0                              MOV     SI,AX
  0024   BA 00 00           R               MOV     DX,SEG _y
  0027   BF 0C 00           R               MOV     DI,OFFSET _y+0
  002A   8E C2                              MOV     ES,DX
  002C   B9 06 00                           MOV     CX,6
  002F   FC                                 CLD
  0030                               L0:
  0030   58                                 POP     AX
  0031   AB                                 STOSW
  0032   E2 FC                              LOOP    L0
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   {
  0000                                      PUBLIC  _main
  0000                                      ASSUME  CS:prog_time,DS:data
  0000                               _main  PROC    FAR
  0000
  0000   1E                                 PUSH    DS
  0001   B8 00 00           R               MOV     AX,data
  0004   8E D8                              MOV     DS,AX
7           y = x;
```

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

```
     0006   BA 00 00          R                MOV     DX,SEG _x
     0009   B8 00 00          R                MOV     AX,OFFSET _x+0
     000C   83 EC 0C                           SUB     SP,12
     000F   96                                 XCHG    AX,SI
     0010   8C DB                              MOV     BX,DS
     0012   8E DA                              MOV     DS,DX
     0014   8B FC                              MOV     DI,SP
     0016   8C D1                              MOV     CX,SS
     0018   8E C1                              MOV     ES,CX
     001A   B9 06 00                           MOV     CX,6
     001D   FC                                 CLD
     001E   F3 A5                              REP MOVSW
     0020   8E DB                              MOV     DS,BX
     0022   8B F0                              MOV     SI,AX
     0024   BA 00 00          R                MOV     DX,SEG _y
     0027   BF 0C 00          R                MOV     DI,OFFSET _y+0
     002A   8E C2                              MOV     ES,DX
     002C   8B F4                              MOV     SI,SP
     002E   8C DA                              MOV     DX,DS
     0030   8C D1                              MOV     CX,SS
     0032   8E D9                              MOV     DS,CX
     0034   B9 06 00                           MOV     CX,6
     0037   FC                                 CLD
     0038   F3 A5                              REP MOVSW
     003A   83 C4 0C                           ADD     SP,12
     003D   8E DA                              MOV     DS,DX
     003F   8B F0                              MOV     SI,AX
   8   }
```

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

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.

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.

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.

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, NEAR jumps with 16-bit displacements are generated by default.

The peephole optimizer, on the other hand, is capable of determining the distance of forward branches, and it will replace NEAR jump instructions with SHORT jump instructions wherever possible.

Tail Merging

A *tail* is a sequence of instructions before an unconditional jump.

When two blocks of code end in identical branches, the peephole optimizer checks if the blocks have the same tail statements. If the blocks do have identical tail statements, the peephole optimizer will replace the first tail with a "goto" the second. For example:

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

Tail merging can take place wherever tails are found, including **if-then-else** and **switch** statements. The compiler does not limit the size of tails that can be merged.

If tail merging would cause an additional branch to be executed, it is not performed when "optimize for time" is specified.

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.)

```
MOV BX,DS: WORD PTR__i[0]
. . .
MOV BX,DS: WORD PTR__i[0]                    ; 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). One to two bytes will be saved by the example strength reduction optimization shown below.

```
MOV BX,DS: WORD PTR__i[0]    ⇒    MOV BX,DS: WORD PTR__i[0]
MOV AX,DS: WORD PTR__i[0]    ⇒    MOV AX,BX
```

Redundant Scale Calculation Elimination

The array index in C must be scaled to its corresponding value in assembly code. For example: In an array of integers, the index value must be doubled. The peephole optimizer removes any redundant scaling. In the code shown below, the second scaling calculation would be removed:

```
MOV    BX,DS: WORD PTR__i[0]
SHL    BX,1
                .
                .
                .
MOV    BX,DS: WORD PTR__i[0]
SHL    BX,1
```

Before the second scaling calculation, the optimizer verifies that the contents of BX and _i have not been changed between the two scaling operations.

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, and redundant scale calculation elimination.

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 four bytes or less, the instructions which are used to push the old stack frame pointer at function entry and restore the frame pointer on exit are not generated.

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 8086/186 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.

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 SEGMENT DATA= idata** to specify the segment 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< emul_env>* and the following error message appears:

```
ERROR:  Could not create
     /usr/hp64000/env/hp<emul_env>/ioconfig.EB
```

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, *cc8086* 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 segment **idata** by default, and static variable definitions without initializers in segment **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 segment **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 segment (**idata**). To do this, place a **# pragma SEGMENT DATA= idata** directive before the initialized variable's external declaration and a **# pragma SEGMENT UNDO** following it. The second pragma merely "undoes" the first pragma. See the "Embedded Systems" chapter for more details on using these pragmas.

Using embedded assembly code with small memory model

Description: For the small memory model, the compiler places all data objects in an assembly language group called **data_const**. When writing embedded assembly code, group-relative accesses MUST be performed instead of segment-relative accesses to static variables and constants. Using segment-relative accesses can cause non-functional code to be produced.

Memory Models

Memory models determine how both segments are to be mapped into memory and the size of pointers. The 8086/186 C compiler provides four memory models, small, compact, medium, and large. The *small memory model* uses fixed segments and 16-bit pointers. The *compact memory model* provides one or more data segments and one code segment. The *medium memory model* uses one or more code segments and one data segment. The *large memory model* provides a flexible number of non-fixed segments and uses 32-bit pointers. Throughout a program, a single memory model must be used; code modules compiled with different memory models cannot be linked together.

Small memory model

The small memory model has two *physical* segments which never change. One is a code segment (CS register does not change). The other is a combined microprocessor stack and DS-relative static data group called **data_const.** (DS, SS, and ES registers are identical and do not change.) This group contains all the data, stack, heap, and constant type segments. This group is placed into a single physical segment at link time. Data and constants are accessed group-relative.

There are no ES-relative static data segments in this model. Both the function and data pointer sizes are 16 bits. Pointer subtraction between function and data pointers will yield unknown results.

Segment groups and classes are discussed in greater detail in your linker manual.

Large memory model

The large memory model may have one or more code segments (CS register may change), one independent stack segment (SS register does not change), zero or one DS-relative static data segment for each C function (DS register may change), and zero or more ES-relative static data segments (ES register may change). Both function and data pointer sizes are 32 bits.

Except for comparisons between two pointers, pointer arithmetic is performed only on the lower 16 bits (the OFFSET part of the SEGMENT:OFFSET address). Operations to compare two pointers are performed using the complete logical address; no translation to a physical address is done.

Functions are considered to be FAR and are called as such (except when a static function is encountered *and* the user has specified the compiler option which says that static functions are to be NEAR).

The last defined static data segment preceding a function is accessed DS-relative. (See **# pragma SEGMENT** and **# pragma DS**.) All other static data segments are accessed ES-relative within that function.

Note

Only one static data segment can be DS-relative per function, but that segment can be different for each function.

Medium Memory Model

The medium memory model may have one or more code segments (the CS register may change) and one data segment (the DS, SS, and ES registers are identical and do not change). The function pointer size is 32 bits, and the data pointer size is 16 bits.

Compact Memory Model

The compact memory model has one fixed code segment (the CS register does not change) and one or more data segments (the DS, SS, and ES registers are not identical and may change). The function pointer size is 16 bits, and the data pointer size is 32 bits.

Segment Names

Segment 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 segments as follows:

- Executable code is placed in the PROG segment (by default, named either **prog_***basename* when using the largeor medium memory model, or **prog** when using the small or compact memory model).
- Static variables are placed in the DATA segment (named **data** by default).
- Constants and string literals are placed in the CONST segment (named **const** by default).

When declaring external data, it is important that the declaration be placed in the segment where the data actually resides. If this is not done, a run-time error may occur when the wrong segment base is used for accessing an external.

All code generated by the compiler is placed in segments with the class name "CODE". Thus the complete name of the default PROG segment is **prog_***basename/CODE*.

If there are multiple declarations for the same symbol within a single file, the compiler checks that the segment in which the symbol is declared is the same in all cases.

Segment name defaults

For large and medium memory model, the compiler allows more than one user program segment. To facilitate easy use of the compiler when user code exceeds 64K bytes (the maximum that can be placed in a single segment), the default PROG segment name is based, in part, on the C source filename. Thus, with large and medium memory model the default PROG segment name is of the form **prog_***basename* where *basename* is the C source file name with the ".c" suffix removed and any illegal characters (for a segment name) changed to underscore (_).

When using the small and compact memory model, the PROG segment name is always **prog.** Since the **SEGMENT** pragma is not valid for small memory model (only one user program segment is allowed) the user cannot change this segment name.

The DATA segment name defaults to **data**. When the "separate initialized data and uninitialized data" option is used, DATA is replaced with IDATA and UDATA which default respectively to **idata** and **udata**.

The CONST segment name defaults to **const.**

Like the PROG segment name, DATA, IDATA, UDATA, and CONST segment names cannot be altered from their defaults when using small and compact memory model. When using large and medium memory model, these segment names can be changed with a **SEGMENT** pragma in the C source.

pragma SEGMENT

Syntax:

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

Note This pragma is only valid for the large, medium, and compact memory model. It is warned and ignored if it is used with the small memory model.

Description

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

In the second form, 20-bit physical addresses are given in place of the segment names causing the subsequent information to be ORG'd starting at the given address. The segment name associated with an ORG'd segment is of the form org*hexaddress*, where *hexaddress* is the physical address of the segment. For example, segment org00012345H is located at 0x12345.

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

```
#pragma SEGMENT 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 segment (const) and since integers "m, n, o" also need to be ORG'd as they are data. Corrected this becomes:

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

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

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

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

In the absence of a **DS** pragma, the DATA segment (or UDATA segment when using "separate initialized and uninitialized data" option) in effect at function entry is the default segment for DS-relative data accesses. (See below for information on the **DS** pragma.)

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

This compiler places all code in the class 'CODE'. No other class names are supported.

The SEGMENT pragma must be placed outside a function body.

Note

 # pragma SEGMENT UNDO is implemented by a one-level-deep stack. That is, only the most recent **SEGMENT** pragma may be "undone" or, said another way, two **# pragma SEGMENT UNDO**s in a row will not undo two **SEGMENT** 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 **SEGMENT-SEGMENT 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.

pragma DS

Syntax:

`#pragma DS segmentName`

Note

This pragma is only valid for the large and compact memory model. It is warned and ignored if it is used with the small and medium memory model.

Description

This pragma specifies that all subsequent functions should arrange to access any data in segment *segmentName*, rather than the default of the current DATA (or UDATA) segment name, using DS-relative addressing. (See the **SEGMENT** pragma regarding default segment names.) If subsequent functions access any static data in segment *segmentName*, their preambles load DS with *segmentName* and use it in accesses. The effect of this is that once a **DS** pragma is used, the DS-relative segment name is fixed until another **DS** pragma is encountered.

RAM and ROM Considerations

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

The C language specifies that, without explicit initialization, static (C *static* or *extern*) variables will be initialized to zero. Declarator initializers allow you to specify initial values other than zero. The following subsections discuss how these variables are initialized in different environments.

No initialized RAM data

There is an "uninitialized data" option for the compiler which prevents initialization to zero of all static variables which have no explicit initialization. Normally, these static variables are specified by the C language to be initialized to zero.

The "uninitialized data" option also causes warning messages to be printed whenever static initializers are used in non-constant declarations. Observe that this option does not prevent the generation of "initialized data" when the user explicitly initializes a static (C *static* or *extern*) variable. By using this option you can verify that your program contains no variables requiring initialization.

The "uninitialized data" option cannot check for the use of a static 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.

RAM data initialized from mass storage

Programs executed in operating systems, in emulation environments, or in simulation environments have a "load time" where initialization can occur. The initial values, or default values of zero, for static variables are therefore written to RAM at load time.

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

The value of this "separate initialized and uninitialized data" option is that it allows the loader to load initialized static data *contiguously* into RAM from the **idata** segment. Also, locations in the **udata** segment can be set to zero in an efficient, contiguous manner, if uninitialized data is to be given default initialization.

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 on explicit initializations of non-constant static data, but places such data in segment **idata** (by default). Static data which is not explicitly initialized is reserved space in segment **udata** (by default), but is not initialized to zero at emulation/simulation load time.

RAM data initialized from ROM

Unlike environments with mass storage, such as in operating systems or emulators, embedded environments have no "load time" and therefore cannot have load time initialization. As an example, when a target system is powered up, the contents of RAM data locations are not defined. However, the C language allows for a "prior to execution" initialization of static variables. To accomplish this initialization, the program's start-up code (**crt0** or **crt1**) can invoke a run time routine (**_initdata()**) to copy initial value data from ROM to RAM for these variables. The "initial value data" ROM tables which **_initdata()** reads are placed in a special series of segments. These segments are named **??DATA1**, **??DATA2**, etc. in segment class *??INIT*. The segment class is used when referencing the segments in the linker command file. The number of segments actually used depends on how much space is needed for the tables.

The default linker command files which are shipped with the compiler are configured such that the "initial value data" tables are not constructed and the run time initialization of static variables is not performed. Only minor modification of the linker command file is needed for the tables to be built by the linker and the **_initdata()** routine to be called from **crt0** (or **crt1**).

Where to load constants

Symbols declared with the **const** type modifier are considered to be ROM locations and are initialized by definition (small memory model differs in this regard, see the next subsection for more detail). For RAM/ROM embedded systems, both program and constants will ultimately reside in ROM and therefore the default segments **prog** and **const** contain ROMable information. In contrast, segments which hold program variables are not ROMable, but instead must be placed in RAM.

RAM and ROM for small memory model

With the small memory model, constants will be placed in a segment named **const**, and static data will be placed in a segment named **data** (**idata** or **udata** if the "separate initialized and uninitialized data" option is on). These four segments (**const**, **data**, **idata**, **udata**) plus the stack and heap segments (**userstack**, **heap**) are placed by the compiler in a group named *data_const*. The total size of this group, after linking, must be no more than 64K bytes; all segments in the group are linked to become a single physical segment. Therefore, although it is possible to position the **const** segment to be placed in

ROM, there must be RAM nearby in the address space to hold the other non-constant segments in the group. If the embedded environment is such that RAM and ROM are are too distant (size of *data_const* would become greater than 64K bytes) then segment **const** must be placed in RAM and initialized at either load time, if it exists, or at run time. Initialization of constants in RAM is done identically to that for "initialized data"; the assembly code produced by the compiler for allocating a constant is the same as that for allocating an "initialized data" variable.

In summary, for small memory model, constants can be placed in ROM if there is RAM nearby to hold data. Otherwise constants must be placed in RAM, along with the data, and then initialized at either load time (emulator, simulator, or operating system environment) or at run time (embedded environment).

Placement of External Declarations

The compiler expects that all external data or constant declarations be explicitly placed in the same named segment in which the data or constant is defined (where storage is allocated). For example, if a static variable *int x* is defined in one file to be in DATA segment **my_data1**, then any *extern int x* declaration MUST be placed in segment **my_data1**. Failure to place external delcarations in their correct segments may result in non-functional code. The compiler uses this segment information to determine if it can or cannot perform a DS-relative access on a given variable or constant.

With small memory model, because the segment names are predefined and not alterable by the user, externals are handled properly without the use of **SEGMENT** pragmas.

Care must be taken when declaring external, initialized data when the "separate initialized and uninitialized data" option is in effect under large memory model. With this option in effect, initialized data definitions will be placed in segment **idata** by default (no **SEGMENT** pragma). However, with this same option, all external data declarations will be placed in segment **udata** by default. The compiler cannot know whether an external variable is initialized or uninitialized and therefore assumes it to be uninitialized and chooses the UDATA default segment name (**udata**). The compiler warns when it makes this assumption. But because this assumption is wrong (external initialized variables are really in **idata**), incorrect code will result.

It is imperative that when using the "separate initialized and uninitialized data" option all external declarations of initialized data be placed in the correct segment as shown in the following example. Note that DATA must be used, instead of IDATA or UDATA, to tell the compiler where an external is located.

File: main.c

```
#pragma SEGMENT DATA=idata
extern int abc;                  /* Tell compiler "abc" is in segment "idata". */
#pragma SEGMENT UNDO

#pragma SEGMENT DATA=udata
extern int def;                  /* Tell compiler "def" is in segment "udata". */
#pragma SEGMENT UNDO

main()
{
   abc++;
   def = abc;
}
```

File: ext_data.c

```
int abc=123;       /* "abc" is allocated space in segment "idata". */
int def;           /* "def" is allocated space in segment "udata". */
```

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 6-1 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 "MOV AX,SI" 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 char    *p = (char *) 0x12340005; /* 0x12340005 is logical address of
                                              I/O port.  0x12345 is physical
                                              address of I/O port.            */
main()
{
        *p = 0;         /* Exactly one write to address 0x12345.  */
        *p;             /* Exactly one read of address 0x12345.   */
}
```

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

Note If the "byte 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   {
        PUBLIC  _main
        ASSUME  CS:%CodeSegment,DS:data
_main   PROC    FAR
%SET(SAVE_ALL_NPX,2)
        PUSH    BP
        MOV     BP,SP
        SUB     SP,4
        PUSH    DS
        MOV     AX,data
        MOV     DS,AX
        PUSH    SI
;S_firstValue is in register SI.
%SET(S_secondValue,-4)
%SET(S_tmp,-2)
    5           int firstValue, secondValue, tmp;
    6
    7           firstValue = io_port;
        MOV     SI,%DS:WORD PTR _io_port[0]
    8           secondValue = io_port;
        MOV     AX,SI
        MOV     SS:WORD PTR [BP+%S_secondValue+0],AX
    9           tmp = firstValue;
        MOV     SS:WORD PTR [BP+%S_tmp+0],SI
   10   }
```

OPTIMIZATION PERFORMED

```
--------------------------------------------------------------------------------
    1   volatile int io_port;
    2
    3   main()
    4   {
        PUBLIC  _main
        ASSUME  CS:%CodeSegment,DS:data
_main   PROC    FAR
%SET(SAVE_ALL_NPX,2)
        PUSH    BP
        MOV     BP,SP
        SUB     SP,4
        PUSH    DS
        MOV     AX,data
        MOV     DS,AX
        PUSH    SI
;S_firstValue is in register SI.
%SET(S_secondValue,-4)
%SET(S_tmp,-2)
    5           int firstValue, secondValue, tmp;
    6
    7           firstValue = io_port;
        MOV     SI,%DS:WORD PTR _io_port[0]
    8           secondValue = io_port;
        MOV     AX,%DS:WORD PTR _io_port[0]
        MOV     SS:WORD PTR [BP+%S_secondValue+0],AX
    9           tmp = firstValue;
        MOV     SS:WORD PTR [BP+%S_tmp+0],SI
   10   }
```

NO OPTIMIZATION PERFORMED

Figure 6-1. "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 8086/186 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	**free**	**malloc**	**rewind**
atexit	**freopen**	**open**	**scanf**
calloc	**fscanf**	**printf**	**setbuf**
close	**fseek**	**putc**	**setvbuf**
fclose	**fsetpos**	**putchar**	**srand**
fflush	**ftell**	**puts**	**strtok**
fgetc	**fwrite**	**rand**	**strtol**
fgetpos	**getc**	**read**	**ungetc**
fgets	**getchar**	**realloc**	**unlink**
fopen	**gets**	**remove**	**vfprintf**
fprintf	**lseek**		**vprintf**
fputc			**write**
fputs			
fread			

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 (plus any register variables which have been allocated), no parameter passing or returned result is allowed, and a return from interrupt is generated at the return point.

If you are using assembly language code, remember that registers which are not used by the compiler as working registers or as register variables are *not* saved at function entry. See page 87 for a list of the compiler's working registers.

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, integer divide-by-zero interrupts are handled by an environment-dependent file which is automatically linked in. Its source (**div_by_0.c**) contains a vector table 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 SEGMENT DATA=0x00

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

#pragma SEGMENT UNDO
#pragma INTERRUPT

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

Note

When using small memory model, the vector table must be coded in assembly in order to specify both the segments and offsets of the interrupt handlers. See the figure in the "Compiler Generated Assembly Code" chapter for an example.

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 cc8086 in the on-line man page.

7

Libraries

Descriptions of the run-time and support libraries.

Four varieties of libraries are provided with the 8086/186 C compiler. Each of these libraries comes in four versions: small memory model, compact memory model, medium memory model, and large memory model. Four versions are provided because you cannot mix memory models within a program. All code must be compiled and linked with the same memory model option.

A check is done at link-time to ensure that all libraries and user-written code have been compiled using the same memory model. This feature eliminates code defects due to mixing the memory models. These defects could be stack misalignment, use of garbage data, and incompatibility of code or data sizes. These code defects would be very hard to find.

A separate version of the math library is provided for use with the 8087. The cc80186 compiler shares the cc8086 libraries.

The four varieties of libraries are:

- **Environment libraries** which contain environment-dependent routines, such as *exit()*, *open()*, *sbrk()*, etc. See the "Environment Dependent Routines" chapter for full details.
- **Run-time libraries** which contain routines required to do real number arithmetic, initializations, run-time debug checks, etc.
- **Support libraries** which contain C functions such as *fopen()*, *getchar()*, *malloc()*, *printf()*, etc.
- **Math libraries** which contain C functions such as *exp()*, *floor()*, *sin()*, etc.

A group of **.h** include files are also provided for use with the various libraries.

The names of the various libraries and the segment names used to locate them by the linker are given below. The names of the libraries for the four memory models are the same, but the directories where they reside on your computer are different.

Table 7-1. Library Names

Library	Library Name	Large Memory Model Segment (PROG, DATA)	Small Memory Model Segment (PROG, DATA)
Environment	env.a	env/CODE, envdata, userstack, heap	prog/CODE, data, userstack, heap
Run-time	lib.a	lib/CODE, libdata	prog/CODE, data
Run-time (8087)	lib87.a	lib/CODE, libdata	prog/CODE, data
Support	libc.a	libc/CODE, libcdata	prog/CODE, data
Math	libm.a	libm/CODE	prog/CODE
Math (8087)	libm87.a	libm/CODE	prog/CODE

Run-Time Library Routines

The run-time library, **lib.a** or **lib87.a**, contains routines used at run-time by the compiler-generated code. The calls to these routines are placed in the assembly code file by the compiler in place of generated assembly code (in-line code). The reasons for using library calls instead of generating in-line code vary from conserving space to minimizing repetition of in-line code to maintenance considerations (the same reasons C functions are used).

The run-time libraries may be called from compiler-generated code and assembly code (including embedded assembly code within the C source). Also, it should be possible to replace any or most of the library routines with your own routines.

The names of all run-time library routines end in _S for small memory model libraries, _C for compact memory model libraries, _M for medium memory model libraries, and _L for large memory model libraries. This is to guarantee that a library routine from one memory model will never be accidentally linked to a call for the other memory model.

The names of all run-time library routines end in _SC for small and compact memory model libraries, and in _LM for large and medium memory model libraries.

See appendices "Small Memory Model Run-time Routines" and "Large Memory Model Run-Time Routines" for descriptions of the interface and functionality of all run-time library routines.

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").

Library Routines Not Provided

Several "standard" C library routines are **not** provided with the 8086/186 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**.

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
*	ADD_F32A_*size*	lib lib87
*	ADD_F32B_*size*	lib lib87
*	ADD_F32C_*size*	lib lib87
*	ADD_F64A_*size*	lib lib87
*	ADD_F64B_*size*	lib lib87
*	ADD_F64C_*size*	lib lib87
*	DEC_F32_*size*	lib lib87
*	DEC_F64_*size*	lib lib87
*	DIV_F32A_*size*	lib lib87
*	DIV_F32B_*size*	lib lib87
*	DIV_F32C_*size*	lib lib87

Index	Definition name	Library
*	DIV_F64A_*size*	lib lib87
*	DIV_F64B_*size*	lib lib87
*	DIV_F64C_*size*	lib lib87
*	DIV_I32A_*size*	lib lib87
*	DIV_I32B_*size*	lib lib87
*	DIV_UI32A_*size*	lib lib87
*	DIV_UI32B_*size*	lib lib87
	DPADD_*size*	lib lib87
	DPDIV_*size*	lib lib87
	DPMUL_*size*	lib lib87
	DPRDIV_*size*	lib lib87

Index	Definition name	Library
*	EQUAL_F32_*size*	lib lib87
*	EQUAL_F64_*size*	lib lib87
	Err_Handler	lib lib87
*	F32_TO_F64_*size*	lib lib87
*	F32_TO_I16_*size*	lib lib87
*	F32_TO_I32_*size*	lib lib87
*	F32_TO_UI16_*size*	lib lib87
*	F32_TO_UI32_*size*	lib lib87
*	F64_TO_F32_*size*	lib lib87
*	F64_TO_I16_*size*	lib lib87
*	F64_TO_I32_*size*	lib lib87
*	F64_TO_UI16_*size*	lib lib87
*	F64_TO_UI32_*size*	lib lib87
*	FAULT_I16_*size*	lib lib87
*	FAULT_I32_*size*	lib lib87
*	FAULT_I8_*size*	lib lib87

Index	Definition name	Library
*	FAULT_PTR_*size*	lib lib87
*	FAULT_UI16_*size*	lib lib87
*	FAULT_UI32_*size*	lib lib87
*	FAULT_UI8_*size*	lib lib87
	FPADD_*size*	lib lib87
	FPDIV_*size*	lib lib87
	FPMUL_*size*	lib lib87
	FPRDIV_*size*	lib lib87
*	I16_TO_F32_*size*	lib lib87
*	I16_TO_F64_*size*	lib lib87
*	I32_TO_F32_*size*	lib lib87
*	I32_TO_F64_*size*	lib lib87
*	INC_F32_*size*	lib lib87
*	INC_F64_*size*	lib lib87
*	LESS_EQ_F32_*size*	lib lib87
*	LESS_EQ_F64_*size*	lib lib87

Index	Definition name	Library
*	LESS_F32_*size*	lib lib87
*	LESS_F64_*size*	lib lib87
	L_1_IO_check_loop	env
	L_2_IO_exit_loop	env
*	MOD_I32A_*size*	lib lib87
*	MOD_I32B_*size*	lib lib87
*	MOD_UI32A_*size*	lib lib87
*	MOD_UI32B_*size*	lib lib87
	MONITOR_MESSAGE	env
*	MUL_F32A_*size*	lib lib87
*	MUL_F32B_*size*	lib lib87
*	MUL_F32C_*size*	lib lib87
*	MUL_F64A_*size*	lib lib87
*	MUL_F64B_*size*	lib lib87
*	MUL_F64C_*size*	lib lib87
*	MUL_I32A_*size*	lib lib87
*	MUL_I32B_*size*	lib lib87

Index	Definition name	Library
*	SUB_F32A_*size*	lib lib87
*	SUB_F32B_*size*	lib lib87
*	SUB_F32C_*size*	lib lib87
*	SUB_F64A_*size*	lib lib87
*	SUB_F64B_*size*	lib lib87
*	SUB_F64C_*size*	lib lib87
	TOP_OF_STACK	env
	UI16_TO_F32_*size*	lib lib87
*	UI16_TO_F64_*size*	lib lib87
*	UI32_TO_F32_*size*	lib lib87
*	UI32_TO_F64_*size*	lib lib87
	USER_ENTRY	env
	USR_STACK	env
	XEnv_86_except	env
	__TOP_OF_HEAP	env
	__USR_HEAP	env
*	___fflush	libc
*	__assert	libc

Index	Definition name	Library
	__bufendtab	env
	__bufsync	libc
*	__clear_fp_status	lib lib87
*	__ctype	libc
	__dbl_to_str	libc
*	__display_message	env
	__doprnt	libc
	__doscan	libc
	__err_handler	lib lib87
*	__error_msg	env
	__exec_funcs	libc
*	__exit	env
	__exit_msg	env
	__filbuf	libc
	__findbuf	libc
	__findiop	libc
	__flsbuf	libc
*	__fp_control	lib
*	__fp_error	libm libm87
*	__fp_status	lib
	__fp_trap	env

Index	Definition name	Library
*	__get_fp_control	lib lib87
*	__get_fp_status	lib lib87
*	__getmem	env
	__hex_NaN	libm libm87
	__hex_NaNf	libm libm87
*	__infinity	libc
*	__init_fp	lib lib87
*	__initdata	env
	__io_bufsiz	env
	__iob	env
	__lastbuf	env
	__lconv_data	libc
	__malloc_init	libc
	__memccpy	libc
*	__open_max	env
	__rand_seed	libc
	__readFile	libc
	__readStr	libc
*	__set_fp_control	lib lib87
	__sibuf	env

Index	Definition name	Library
	__smbuf	env
	__sobuf	env
*	__startup	env
	__stdbuf	env
	__swrite	libc
	__top_of_func_stack	libc
	__wrtchk	libc
	__xflsbuf	libc
*	_abs	libc
	_abs_out_adrs	env
*	_acos	libm libm87
*	_asin	libm libm87
*	_atan	libm libm87
*	_atan2	libm libm87
*	_atexit	libc
*	_atof	libc
*	_atoi	libc
*	_atol	libc
*	_bsearch	libc

Index	Definition name	Library
*	_calloc	libc
*	_ceil	libm libm87
*	_clear_screen	env
*	_clearerr	libc
*	_close	env
*	_cos	libm libm87
*	_cosh	libm libm87
	_count	env
	_data_buff	env
	_data_ptr	env
*	_div	libc
*	_errno	libc
*	_exec_cmd	env
*	_exit	env
*	_exp	libm libm87
*	_fabs	libm libm87
*	_fclose	libc
*	_feof	libc
*	_ferror	libc

Index	Definition name	Library
*	_fflush	libc
*	_fgetc	libc
*	_fgetpos	libc
*	_fgets	libc
*	_floor	libm libm87
*	_fmod	libm libm87
*	_fopen	libc
*	_fprintf	libc
*	_fputc	libc
*	_fputs	libc
*	_fread	libc
*	_free	libc
*	_frem	libm libm87
*	_freopen	libc
*	_frexp	libm libm87
*	_fscanf	libc
*	_fseek	libc
*	_fsetpos	libc
*	_ftell	libc

Index	Definition name	Library
*	_fwrite	libc
*	_getc	libc
*	_getchar	libc
*	_gets	libc
*	_initsimio	env
*	_isalnum	libc
*	_isalpha	libc
*	_iscntrl	libc
*	_isdigit	libc
*	_isgraph	libc
*	_islower	libc
*	_isprint	libc
*	_ispunct	libc
*	_isspace	libc
*	_isupper	libc
*	_isxdigit	libc
*	_kill	env
*	_labs	libc
*	_ldexp	libm libm87
*	_ldiv	libc

Index	Definition name	Library
*	_localeconv	libc
*	_log	libm libm87
*	_log10	libm libm87
*	_longjmp	libc
*	_lseek	env
*	_malloc	libc
*	_mblen	libc
*	_mbstowcs	libc
*	_mbtowc	libc
*	_memchr	libc
*	_memcmp	libc
*	_memcpy	libc
*	_memmove	libc
*	_memset	libc
*	_modf	libm libm87
*	_open	env
	_open_file	env
*	_perror	libc
*	_pos_cursor	env

Index	Definition name	Library
*	_pow	libm libm87
*	_printf	libc
*	_putc	libc
*	_putchar	libc
*	_puts	libc
*	_qsort	libc
*	_rand	libc
*	_read	env
*	_realloc	libc
*	_remove	libc
*	_rewind	libc
*	_sbrk	env
*	_scanf	libc
*	_setbuf	libc
*	_setjmp	libc
*	_setlocale	libc
*	_setvbuf	libc
*	_sin	libm libm87
*	_sinh	libm libm87

Index	Definition name	Library
*	_sprintf	libc
*	_sqrt	libm libm87
*	_srand	libc
*	_sscanf	libc
*	_strcat	libc
*	_strchr	libc
*	_strcmp	libc
*	_strcoll	libc
*	_strcpy	libc
*	_strcspn	libc
*	_strerror	libc
*	_strlen	libc
*	_strncat	libc
*	_strncmp	libc
*	_strncpy	libc
*	_strpbrk	libc
*	_strrchr	libc
*	_strspn	libc
*	_strstr	libc
*	_strtod	libc

Index	Definition name	Library
*	_strtok	libc
*	_strtol	libc
*	_strtoul	libc
*	_strxfrm	libc
	_systemio_buf	env
*	_tan	libm libm87
*	_tanh	libm libm87
*	_tolower	libc
*	_toupper	libc
*	_ungetc	libc
*	_unlink	env
*	_vfprintf	libc
*	_vprintf	libc
*	_vsprintf	libc
	_wait_for_io	env
*	_wcstombs	libc
*	_wctomb	libc
*	_write	env

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 file **libm.a**. All other functions are found in the support library archive file **libc.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 coprocessor 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);

void _init_fp (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 8087 NPX. Therefore, *_fp_error* is referenced only when the 8086/186 library **libm.a**) is loaded.

_fp_error composes the return value defined by the IEEE Floating Point Standard 754 (see the "Behavior of Math Library Functions" chapter) and returns the value if trapping does not take place. The trapping decision is handled by a piece of common code in the run-time library. This code inspects a global control flag to see if the trap bit associated with the current exception is set. If the bit is set, an error message is composed and control passes to the monitor program so that the message can be displayed. If the bit is not set, then a global status flag is updated to reflect the exception that just occurred and processing continues.

The following functions can be used to inspect and set the global control flag and the global status flag:

_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.

_init_fp resets the 8087 by executing the FINIT instruction (if **lib87.a** is being used) and clears the global status flag and the global control flag.

The 8086/186 libraries always perform operations in double precision and round to nearest. By default, trapping is enabled on all floating-point exceptions except inexact results. The following macro disables trapping:

NOTRAP	Disable all traps.

The remaining macros may be OR'ed together to form *mode* when invoking *_set_fp_control*. (Do not OR them with NOTRAP.)

INEXACT	Trap on inexact result.
DIVZERO	Trap on division by zero.
UNDERFLOW	Trap on underflow.
OVERFLOW	Trap on overflow.
OPERROR	Trap on operand error.
PLOSS	Trap on loss of precision (applies to 8086/186—not 8087—math libraries).

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*.

INEXACT
DIVZERO
UNDERFLOW
OVERFLOW
OPERROR
PLOSS

When using the 8087 chip, the control word contains some additional information. The 8087 allows you to control trapping, precision, infinity, and rounding behaviors.

The following macros can be used to select the 8087 behavior that is desired:

Precision:

SGLPREC	Single precision (32-bit floating point number).
DBLPREC	Double precision (64-bit floating point number).
EXTPREC	Extended precision (80-bit floating point number).

Infinity:

PROJECTIVE	Infinity is unsigned.
AFFINE	Distinguish + infinity from –infinity.

Rounding:

RNDNEAR	Round towards the "nearest" number.
RNDNEGINF	Round towards negative infinity.
RNDPOSINF	Round towards positive infinity.
RNDZERO	Round towards zero.

Trapping:

DENORM_OP	Trap when a denormalized operand is encountered.

Select exactly one macro each from the precision, rounding, and infinity categories every time that *_set_fp_control* is called. Any number of trapping macros can be selected.

Note that an OPERROR trap can occur when a 8087 floating point register is used before it is initialized.

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.

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.

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. 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 $-\pi/2$ to $\pi/2$.

Acos returns the arccosine of *x*, in the range 0 to π.

Atan returns the arctangent of *x*, in the range $-\pi/2$ to $\pi/2$.

Atan2 returns the arctangent of y / x, in the range $-\pi$ 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 (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.

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.

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. 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 segment name **env**. This segment name may be used just as any other segment name would be (for example, in **SEGMENT** pragmas). See the on-line man pages for a complete description of the cc8086 and cc80186 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.

Program Setup

Two program setup routines are provided with the 8086/186 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. 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**.

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 **_exit_msg()**.

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()** (via **_exit_msg()**) 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.

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 8087 code with **crt0.o**.
flinkcom.k	Links program containing 8087 code with **crt1.o**.

Since C assumes that **stdin**, **stdout**, and **stderr** are opened prior to **main()** being called, cc8086 automatically uses the **iolinkcom.k** (or **fiolinkcom.k**) linker command file. To link with **crt1.o** instead, use the cc8086 "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 8087" (**-f**) option, **fiolinkcom.k** or **flinkcom.k** will be used instead of **iolinkcom.k** or **linkcom.k**. These linker command files substitute **lib87.a** for **lib.a** and **libm87.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 to four configuration files are provided for each supported emulator:

ioconfig.EA For programs linked with crt0.o.

config.EA For programs linked with crt1.o.

fioconfig.EA For programs containing 8087 code and linked with crt0.o.

fconfig.EA For programs containing 8087 code and 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.

Configuration files **fioconfig.EA** and **fconfig.EA** are supplied only for those emulation environments which support the 8087. These configuration files should be used whenever the program contains 8087 code.

Memory Map

Notice that each memory model has its own memory map. Check figures 8-3 and 8-4 (figures 8-5 and 8-6 for HP 647xx emulation environments) to find out where the segments are placed for a particular memory model. The segment ordering is specified by the default linker command files **iolinkcom.k** and **linkcom.k** (**fiolinkcom.k** and **flinkcom.k** when using the 8087). The memory map is defined by the provided emulator configuration files **ioconfig.EA** and **config.EA** (**fioconfig.EA** and **fconfig.EA** when using the 8087). Because emulator configuration files map memory for absolute code located by the linker, modifications to the default linker command files will usually necessitate modifications to the emulator configuration as well.

Note

When using small memory model with the run-time error checking option turned on, **no** user code (PROG Segment **prog/CODE**) or data (DATA Segment **data**) should be placed where OFFSET = 0000. The NULL pointer is defined to be 0000. If you store code or data at OFFSET 0000, the address of that code or data will be confused with the NULL pointer.

Note that the small memory model map has the PROG and DATA segments beginning at 80002H and 10002H to avoid this situation.

00000H (emul ROM) 003FFH	SEGMENT interrupt (1K)	Space reserved for interrupt vectors.
	• • •	
10000H (emul RAM) 1F3FFH	SEGMENT envdata SEGMENT libdata SEGMENT libcdata SEGMENT data SEGMENT idata SEGMENT udata SEGMENT heap (4K) SEGMENT userstack (61K)	Environment-dependent data Run-time library data Support library data Default for user data Initialized user data Uninitialized user data System dynamic pool System stack
	• • •	
80000H (emul ROM) 8FFFFH	CLASS CODE SEGMENT libcconst SEGMENT libmconst CLASS ??INIT SEGMENT mm_check SEGMENT const (64K)	All code space Support library constants Math library constants Initialized-data tables Memory model check Default for user constants
FFFFFH	• • •	

Figure 8-1. Default Memory Map for Large Memory Model

00000H (emul ROM) 003FFH	SEGMENT interrupt (1K)	Space reserved for interrupt vectors.
	• • •	
10002H (emul RAM) 1F3FFH	SEGMENT data SEGMENT idata SEGMENT udata SEGMENT heap (4K) SEGMENT userstack SEGMENT const SEGMENT envdata (61K)	Library & default user data Initialized user data Uninitialized user data System dynamic pool System dynamic stack User constants INITDATA data
	• • •	
80002H (emul ROM) 8FFFFH	CLASS CODE CLASS ??INIT SEGMENT mm_check (64K)	All code space Initialized-data tables Memory model check
  FFFFFH	• • •	

Figure 8-2. Default Memory Map for Small Memory Model

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 (in file **getmem.c**) 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. This routine is similar to the host operating system **sbrk()** function.

After the **_getmem()** function is rewritten and compiled, the new **getmem.o** object file should be loaded before the **env.a** library, or be used (with ar86) to replace the existing **getmem** object module in the **env.a** library. Refer to the "Getting Started" chapter for an automated way to rebuild the **env.a** library using the *make* utility.

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 ar86) to replace the existing **systemio** object module in the **env.a** library. Refer to the "Getting Started" chapter for an automated way to rebuild the **env.a** library using the *make* utility.

Environment-Dependent I/O Functions

The remainder of this chapter describes the I/O library functions which are dependent on the emulator execution environments. 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 via **_exit_msg()**, 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 SEGMENT 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 accordingly.

[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^{16}-2 bytes.

Exceeded parameter passing space. This error occurs when there is too much parameter storage. The limit is 2^{16}-2 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.)

Near function < identifier> called across segments. A call to a static function in a different segment has been attempted with the cc8086 "near calls" option specified.

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 segments.

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 segment and in an absolute program segment (defined with the SEGMENT 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 segment 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 segment (prog, data, or const) and they must all be either defined in the same module or defined externally.

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^{16}-2 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 cc8086 "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 cc8086 "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** segment. If the variable is initialized, it is very important to place the variable in the correct segment (**idata**). To do this, use a **# pragma SEGMENT DATA= idata** before the external declaration to name the initialized data segment. 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 cc8086 "generate additional warnings" option is specified.

Mixing extern declaration of < identifier> with near calls. This warning message is enabled when the cc8086 "NEAR calls" compiler option is specified and a function is declared as **extern** and later as **static**. The resulting symbol is changed from **extern** to **static** in midstream, which may result in incorrect "NEAR" calls to a "FAR" function. Remember that the **extern** declaration may be implicit.

Mixing function pointer < identifier> with near calls.
This warning message is enabled when the cc8086 "near calls" option is specified and function pointers are declared.

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 cc8086 "generate additional warnings" option is specified.

Test expression is an assignment. This warning message is enabled when the cc8086 "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:

Flt Pt Invalid Operation. This error occurs when an operand is invalid for the operation performed. Examples include:

- 0 * Infinity.
- (+ Infinity) + (–Infinity).
- 0/0 or Infinity/Infinity.
- Comparison between NaN and any other value.
- Floating point register variable is read without having been initialized (8087 only).

Flt Pt Overflow. This error occurs when the result of an operation is too large to be represented in the destination format.

Flt Pt Underflow. This 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.

Flt Pt Divide by Zero. This error occurs when attempting to divide a non-zero value by zero. (Zero divided by zero is an invalid operation error.)

Flt Pt Imprecise. This error occurs when the result requires rounding. Due to the high probability of rounding, this trap is typically disabled.

Flt Pt Significance Loss. This error occurs when precision is lost during the reduction of large arguments in the trigonometric functions.

Flt Pt Denormal This error occurs after an operation is attempted on a denormal number (8087 only).

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 may be truncated, if necessary, to 12 characters after 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. This field may be replaced with "max" if the message won't fit on the status line.

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_msg()** library routine and is:

```
Program end, returned <arg>
```

Where <*arg*> is either the value returned by **main()** or the argument passed to an explicit call to **exit()**.

If an integer divide by zero is attempted, the program will terminate with the following message displayed:

```
Integer divide by zero
```

11

Run-Time Routines

Descriptions of run-time routines.

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.

The names of some run-time routines have changed between versions of the compiler; for example, many large model routines were renamed from _L to _LM when support for the medium memory model was added.

The following conventions are followed for this appendix:

< *size*>	_S or _SC for the small memory model, _L or _LM for the large memory model, _C or _SC for the compact memory model, _M or _LM for the medium model.
< *pointer size*>	16 for the small memory model, 32 for the large memory model.
DXAX	32-bit pseudo-register consisting of registers DX and AX with DX holding the most significant word.
ESDI	32-bit pseudo-register consisting of registers ES and DI with ES holding the segment and DI holding the offset.
F64	Double (8 bytes).
F32	Float (4 bytes).
UI32	Unsigned Long (4 bytes).
I32	Signed Long (4 bytes).
UI16	Unsigned Integer (2 bytes).
I16	Signed Integer (2 bytes).

()	Indirection. For example, (DI) represents the memory location pointed to by register DI.
PARM0	Last parameter pushed on the stack.
PARM1	If present, this parameter is pushed on the stack just prior to PARM0. These parameters may be four or eight bytes in size, depending on the specific library routine. Some routines do not use the stack to pass parameters.

Figures 9-1 through 9-4 can be found at the end of this appendix.

Conversion Routines

F64_TO_F32< *size*>

Converts a 64-bit floating point value to a 32-bit floating point value by rounding to nearest. A zero is returned for a denormal 64-bit floating point value.

Input:	F64 in PARM0 (8 bytes).
Output:	F32 in register DXAX.
Registers Destroyed:	BX, CX.
Side Effects:	PARM0 is deallocated by this routine via RET 8.
Synopsis:	DXAX ← cast PARM0.
Stack Upon Entry:	Figure 9-1.

F32_TO_F64< *size*>

Converts a 32-bit floating point value to a 64-bit floating point value. A zero is returned for a denormal 32-bit floating point value. The additional mantissa bits of the 64-bit floating point value are always returned zero, even when converting an NaN.

Input:	F32 in register DXAX.
Output:	F64 in PARM0 (8 bytes).
Registers Destroyed:	BX, CX, DI.
Side Effects:	None.
Synopsis:	PARM0 ← cast DXAX.
Stack Upon Entry:	Figure 9-1.

F64_TO_UI32*< size>*

Converts a 64-bit floating point value to a 32-bit unsigned integer by truncation. Floating point values that cannot be represented by a 32-bit unsigned integer return **0x80000000**.

Input:	F64 in PARM0 (8 bytes).
Output:	UI32 in register DXAX.
Registers Destroyed:	BX, CX, DI.
Side Effects:	PARM0 is deallocated by this routine via RET 8.
Synopsis:	DXAX ←cast PARM0.
Stack Upon Entry:	Figure 9-1.

UI32_TO_F64*< size>*

Converts a 32-bit unsigned integer to a 64-bit floating point value.

Input:	UI32 in register DXAX.
Output:	F64 in PARM0 (8 bytes).
Registers Destroyed:	AX, BX, CX, DX, DI.
Side Effects:	None.
Synopsis:	PARM0 ←cast DXAX.
Stack Upon Entry:	Figure 9-1.

F64_TO_UI16< *size*>

Converts a 64-bit floating point value to a 16-bit unsigned integer by truncation. Floating point values that cannot be represented by a 16-bit unsigned integer return **0x8000**.

Input:	F64 in PARM0 (8 bytes).
Output:	UI16 in register AX.
Registers Destroyed:	BX, CX, DX, DI.
Side Effects:	PARM0 is deallocated by this routine via RET 8.
Synopsis:	AX $\leftarrow$ cast PARM0.
Stack Upon Entry:	Figure 9-1.

UI16_TO_F64< *size*>

Converts a 16-bit unsigned integer to a 64-bit floating point value.

Input:	UI16 in register AX.
Output:	F64 in PARM0 (8 bytes).
Registers Destroyed:	AX, BX, CX, DX, DI.
Side Effects:	None.
Synopsis:	PARM0 $\leftarrow$ cast AX.
Stack Upon Entry:	Figure 9-1.

F64_TO_I32< *size*>

Converts a 64-bit floating point value to a 32-bit signed integer by truncation. Floating point values that cannot be represented by a 32-bit signed integer return **0x80000000**.

Input:	F64 in PARM0 (8 bytes).
Output:	I32 in register DXAX.
Registers Destroyed:	BX, CX, DI.
Side Effects:	PARM0 is deallocated by this routine via RET 8.
Synopsis:	DXAX ← cast PARM0.
Stack Upon Entry:	Figure 9-1.

I32_TO_F64< *size*>

Converts a 32-bit signed integer to a 64-bit floating point value.

Input:	I32 in register DXAX.
Output:	F64 in PARM0 (8 bytes).
Registers Destroyed:	AX, BX, CX, DX, DI.
Side Effects:	None.
Synopsis:	PARM0 ← cast DXAX.
Stack Upon Entry:	Figure 9-1.

F64_TO_I16< *size*>

Converts a 64-bit floating point value to a 16-bit signed integer by truncation. Floating point values that cannot be represented by a 16-bit signed integer return **0x8000**.

Input:	F64 in PARM0 (8 bytes).
Output:	I16 in register AX.
Registers Destroyed:	BX, CX, DX, DI.
Side Effects:	PARM0 is deallocated by this routine via RET 8.
Synopsis:	AX ← cast PARM0.
Stack Upon Entry:	Figure 9-1.

I16_TO_F64< *size*>

Converts a 16-bit signed integer to a 64-bit floating point value.

Input:	I16 in register AX.
Output:	F64 in PARM0 (8 bytes).
Registers Destroyed:	AX, BX, CX, DX, DI.
Side Effects:	None.
Synopsis:	PARM0 ← cast AX.
Stack Upon Entry:	Figure 9-1.

F32_TO_UI32*< size>*

Converts a 32-bit floating point value to a 32-bit unsigned integer by truncation. Floating point values that cannot be represented by a 32-bit unsigned integer return **0x80000000**.

Input:	F32 in register DXAX.
Output:	UI32 in register DXAX.
Registers Destroyed:	BX, CX, DI.
Side Effects:	None.
Synopsis:	DXAX ← cast DXAX.

UI32_TO_F32*< size>*

Converts a 32-bit unsigned integer to a 32-bit floating point value by rounding to nearest.

Input:	UI32 in register DXAX.
Output:	F32 in register DXAX.
Registers Destroyed:	BX, CX, DI.
Side Effects:	None.
Synopsis:	DXAX ← cast DXAX.

F32_TO_UI16*< size>*

Converts a 32-bit floating point value to a 16-bit unsigned integer by truncation. Floating point values that cannot be represented by a 16-bit unsigned integer return **0x8000**.

Input:	F32 in register DXAX.
Output:	UI16 in register AX.
Registers Destroyed:	CX, DX.
Side Effects:	None.
Synopsis:	AX ←cast DXAX.

UI16_TO_F32*< size>*

Converts a 16-bit unsigned integer to a 32-bit floating point value.

Input:	UI16 in register AX.
Output:	F32 in register DXAX.
Registers Destroyed:	None.
Side Effects:	None.
Synopsis:	DXAX ←cast AX.

F32_TO_I32< *size*>

Converts a 32-bit floating point value to a 32-bit signed integer by truncation. Floating point values that cannot be represented by a 32-bit signed integer return **0x80000000**.

Input:	F32 in register DXAX.
Output:	I32 in register DXAX.
Registers Destroyed:	BX, CX, DI.
Side Effects:	None.
Synopsis:	DXAX ←cast DXAX.

I32_TO_F32< *size*>

Converts a 32-bit signed integer to a 32-bit floating point value by rounding to nearest.

Input:	I32 in register DXAX.
Output:	F32 in register DXAX.
Registers Destroyed:	BX, CX, DI.
Side Effects:	None.
Synopsis:	DXAX ←cast DXAX.

F32_TO_I16< *size*>

Converts a 32-bit floating point value to a 16-bit signed integer by truncation. Floating point values that cannot be represented by a 16-bit signed integer return **0x8000**.

Input:	F32 in register DXAX.
Output:	I16 in register AX.
Registers Destroyed:	CX, DX.
Side Effects:	None.
Synopsis:	AX ←cast DXAX.

I16_TO_F32< *size*>

Converts a 16-bit signed integer to a 32-bit floating point value.

Input:	I16 in register AX.
Output:	F32 in register DXAX.
Registers Destroyed:	None.
Side Effects:	None.
Synopsis:	DXAX ←cast AX.

Floating Point Addition Routines

ADD_F64A< *size*>

Adds two 64-bit floating point values, returning a 64-bit floating point value.

Input:	F64 addend in PARM1 (8 bytes). F64 addor in PARM0 (8 bytes).
Output:	F64 result in PARM1 (8 bytes).
Registers Destroyed:	AX, BX, CX, DX, DI.
Side Effects:	PARM0 is deallocated by this routine via RET 8.
Synopsis:	PARM1 ← PARM1 + PARM0.
Stack Upon Entry:	Figure 9-2.

ADD_F64B< *size*>

Adds two 64-bit floating point values, returning a 64-bit floating point value in two places.

Input:	Pointer to F64 addend/result in DI. F64 addor in PARM0 (8 bytes).
Output:	F64 result in memory location pointed to by DI. F64 result in PARM0 (8 bytes).
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	None.
Synopsis:	(DI),PARM0 ←(DI) + PARM0.
Stack Upon Entry:	Figure 9-1.

ADD_F64C< *size*>

Adds two 64-bit floating point values, returning a 64-bit floating point value.

Input:	Pointer to F64 addend/result in DI. F64 addor in PARM0 (8 bytes).
Output:	F64 result in memory location pointed to by DI.
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	PARM0 is deallocated by this routine via RET 8.
Synopsis:	(DI) ←(DI) + PARM0.
Stack Upon Entry:	Figure 9-1.

INC_F64*< size>*

Adds 1.0 to a 64-bit floating point value, returning two 64-bit floating point values; the original value and the incremented value.

Input:	Pointer to F64 source/result operand in DI.
Output:	Original F64 source value (pointed to by DI) in PARM0 (8 bytes). F64 result in memory location pointed to by DI.
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	None.
Synopsis:	PARM0 $\leftarrow$(DI). (DI) $\leftarrow$(DI) + 1.0.
Stack Upon Entry:	Figure 9-1.

ADD_F32A*< size>*

Adds two 32-bit floating point values, returning a 32-bit floating point value.

Input:	F32 addend in PARM1 (4 bytes). F32 addor in PARM0 (4 bytes).
Output:	F32 result in PARM1 (4 bytes).
Registers Destroyed:	AX, BX, CX, DX, DI.
Side Effects:	PARM0 is deallocated by this routine via RET 4.
Synopsis:	PARM1 $\leftarrow$PARM1 + PARM0.
Stack Upon Entry:	Figure 9-4.

ADD_F32B< *size*>

Adds two 32-bit floating point values, returning a 32-bit floating point value in two places.

Input:	Pointer to F32 addend/result in DI. F32 addor in PARM0 (4 bytes).
Output:	F32 result in memory location pointed to by DI. F32 result in PARM0 (4 bytes).
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	None.
Synopsis:	(DI),PARM0 ← (DI) + PARM0.
Stack Upon Entry:	Figure 9-3.

ADD_F32C< *size*>

Adds two 32-bit floating point values, returning a 32-bit floating point value.

Input:	Pointer to F32 addend/result in DI. F32 addor in PARM0 (4 bytes).
Output:	F32 result in memory location pointed to by DI.
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	PARM0 is deallocated by this routine via RET 4.
Synopsis:	(DI) ← (DI) + PARM0.
Stack Upon Entry:	Figure 9-3.

INC_F32< *size*>

Adds 1.0 to a 32-bit floating point value, returning two 32-bit floating point values: the original value; the incremented value.

Input:	Pointer to F32 source/result operand in DI.
Output:	Original F32 source value (pointed to by DI) in PARM0 (4 bytes). F32 result in memory location pointed to by DI.
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	None.
Synopsis:	PARM0 $\leftarrow$(DI). (DI) $\leftarrow$(DI) + 1.0.
Stack Upon Entry:	Figure 9-3.

Floating Point Subtraction Routines

SUB_F64A< *size*>

Subtracts a 64-bit floating point value from another 64-bit floating point value, returning a 64-bit floating point value.

Input:	F64 minuend in PARM1 (8 bytes). F64 subtrahend in PARM0 (8 bytes).
Output:	F64 result in PARM1 (8 bytes).
Registers Destroyed:	AX, BX, CX, DX, DI.
Side Effects:	PARM0 is deallocated by this routine via RET 8.
Synopsis:	PARM1 ←PARM1 – PARM0.
Stack Upon Entry:	Figure 9-2.

SUB_F64B< *size*>

Subtracts a 64-bit floating point value from another 64-bit floating point value, returning a 64-bit floating point value in two places.

Input:	Ptr to F64 minuend/result in DI. F64 subtrahend in PARM0 (8 bytes).
Output:	F64 result in memory location pointed to by DI. F64 result in PARM0 (8 bytes).
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	None.
Synopsis:	(DI),PARM0 ←(DI) – PARM0.
Stack Upon Entry:	Figure 9-1.

SUB_F64C*< size>*

Subtracts a 64-bit floating point value from another 64-bit floating point value, returning a 64-bit floating point value.

Input:	Pointer to F64 minuend/result in DI. F64 subtrahend in PARM0 (8 bytes).
Output:	F64 result in memory location pointed to by DI.
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	PARM0 is deallocated by this routine via RET 8.
Synopsis:	(DI) ←(DI) – PARM0.
Stack Upon Entry:	Figure 9-1.

DEC_F64*< size>*

Subtracts 1.0 from a 64-bit floating point value, returning two 64-bit floating point values; the original value and the decremented value.

Input:	Pointer to F64 source/result operand in DI.
Output:	Original F64 source value (pointed to by DI) in PARM0 (8 bytes). F64 result in memory location pointed to by DI.
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	None.
Synopsis:	PARM0 ←(DI). (DI) ←(DI) – 1.0.
Stack Upon Entry:	Figure 9-1.

SUB_F32A*<size>*

Subtracts a 32-bit floating point value from another 32-bit floating point value, returning a 32-bit floating point value.

Input:	F32 minuend in PARM1 (4 bytes). F32 subtrahend in PARM0 (4 bytes).
Output:	F32 result in PARM1 (4 bytes).
Registers Destroyed:	AX, BX, CX, DX, DI.
Side Effects:	PARM0 is deallocated by this routine via RET 4.
Synopsis:	PARM1 ← PARM1 – PARM0.
Stack Upon Entry:	Figure 9-4.

SUB_F32B*<size>*

Subtracts a 32-bit floating point value from another 32-bit floating point value, returning a 32-bit floating point value in two places.

Input:	Pointer to F32 minuend/result in DI. F32 subtrahend in PARM0 (4 bytes).
Output:	F32 result in memory location pointed to by DI. F32 result in PARM0 (4 bytes).
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	None.
Synopsis:	(DI),PARM0 ← (DI) – PARM0.
Stack Upon Entry:	Figure 9-3.

SUB_F32C*< size>*

Subtracts a 32-bit floating point value from another 32-bit floating point value, returning a 32-bit floating point value.

Input:	Pointer to F32 minuend/result in DI. F32 subtrahend in PARM0 (4 bytes).
Output:	F32 result in memory location pointed to by DI.
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	PARM0 is deallocated by this routine via RET 4.
Synopsis:	(DI) ←(DI) – PARM0.
Stack Upon Entry:	Figure 9-3.

DEC_F32*< size>*

Subtracts 1.0 from a 32-bit floating point value, returning two 32-bit floating point values; the original value and the decremented value.

Input:	Pointer to F32 source/result operand in DI.
Output:	Original F32 source value (pointed to by DI) in PARM0 (4 bytes). F32 result in memory location pointed to by DI.
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	None.
Synopsis:	PARM0 ←(DI). (DI) ←(DI) – 1.0.
Stack Upon Entry:	Figure 9-3.

Floating Point Multiplication Routines

MUL_F64A< *size*>

Multiplies two 64-bit floating point values, returning a 64-bit floating point value.

Input:	F64 multiplicand in PARM1 (8 bytes). F64 multiplier in PARM0 (8 bytes).
Output:	F64 result in PARM1 (8 bytes).
Registers Destroyed:	AX, BX, CX, DX, DI.
Side Effects:	PARM0 is deallocated by this routine via RET 8.
Synopsis:	PARM1 ← PARM1 ∗PARM0.
Stack Upon Entry:	Figure 9-2.

MUL_F64B< *size*>

Multiplies two 64-bit floating point values, returning a 64-bit floating point value in two places.

Input:	Pointer to F64 multiplicand/result in DI. F64 multiplier in PARM0 (8 bytes).
Output:	F64 result in memory location pointed to by DI. F64 result in PARM0 (8 bytes).
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	None.
Synopsis:	(DI),PARM0 ← (DI) ∗PARM0.
Stack Upon Entry:	Figure 9-1.

MUL_F64C< *size*>

Multiplies two 64-bit floating point values, returning a 64-bit floating point value.

Input:	Pointer to F64 multiplicand/result in DI. F64 multiplier in PARM0 (8 bytes).
Output:	F64 result in memory location pointed to by DI.
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	PARM0 is deallocated by this routine via RET 8.
Synopsis:	(DI) $\leftarrow$(DI) $*$PARM0.
Stack Upon Entry:	Figure 9-1.

MUL_F32A< *size*>

Multiplies two 32-bit floating point values, returning a 32-bit floating point value.

Input:	F32 multiplicand in PARM1 (4 bytes). F32 multiplier in PARM0 (4 bytes).
Output:	F32 result in PARM1 (4 bytes).
Registers Destroyed:	AX, BX, CX, DX, DI.
Side Effects:	PARM0 is deallocated by this routine via RET 4.
Synopsis:	PARM1 $\leftarrow$PARM1 $*$PARM0.
Stack Upon Entry:	Figure 9-4.

MUL_F32B*< size>*

Multiplies two 32-bit floating point values, returning a 32-bit floating point value in two places.

Input:	Pointer to F32 multiplicand/result in DI. F32 multiplier in PARM0 (4 bytes).
Output:	F32 result in memory location pointed to by DI. F32 result in PARM0 (4 bytes).
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	None.
Synopsis:	(DI),PARM0 ←(DI) ∗PARM0.
Stack Upon Entry:	Figure 9-3.

MUL_F32C*< size>*

Multiplies two 32-bit floating point values, returning a 32-bit floating point value.

Input:	Pointer to F32 multiplicand/result in DI. F32 multiplier in PARM0 (4 bytes).
Output:	F32 result in memory location pointed to by DI.
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	PARM0 is deallocated by this routine via RET 4.
Synopsis:	(DI) ←(DI) ∗PARM0.
Stack Upon Entry:	Figure 9-3.

Floating Point Division Routines

DIV_F64A< *size*>

Divides a 64-bit floating point value by another 64-bit floating point value, returning a 64-bit floating point value.

Input:	F64 dividend in PARM1 (8 bytes). F64 divisor in PARM0 (8 bytes).
Output:	F64 result in PARM1 (8 bytes).
Registers Destroyed:	AX, BX, CX, DX, DI.
Side Effects:	PARM0 is deallocated by this routine via RET 8.
Synopsis:	PARM1 ← PARM1 / PARM0.
Stack Upon Entry:	Figure 9-2.

DIV_F64B< *size*>

Divides a 64-bit floating point value by another 64-bit floating point value, returning a 64-bit floating point value in two places.

Input:	Pointer to F64 dividend/result in DI. F64 divisor in PARM0 (8 bytes).
Output:	F64 result in memory location pointed to by DI. F64 result in PARM0 (8 bytes).
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	None.
Synopsis:	(DI),PARM0 ← (DI) / PARM0.
Stack Upon Entry:	Figure 9-1.

DIV_F64C*<size>*

Divides a 64-bit floating point value by another 64-bit floating point value, returning a 64-bit floating point value.

Input:	Pointer to F64 dividend/result in DI. F64 divisor in PARM0 (8 bytes).
Output:	F64 result in memory location pointed to by DI.
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	PARM0 is deallocated by this routine via RET 8.
Synopsis:	(DI) ←(DI) / PARM0.
Stack Upon Entry:	Figure 9-1.

DIV_F32A*<size>*

Divides a 32-bit floating point value by another 32-bit floating point value, returning a 32-bit floating point value.

Input:	F32 dividend in PARM1 (4 bytes). F32 divisor in PARM0 (4 bytes).
Output:	F32 result in PARM1 (4 bytes).
Registers Destroyed:	AX, BX, CX, DX, DI.
Side Effects:	PARM0 is deallocated by this routine via RET 4.
Synopsis:	PARM1 ←PARM1 / PARM0.
Stack Upon Entry:	Figure 9-4.

DIV_F32B*< size>*

Divides a 32-bit floating point value by another 32-bit floating point value, returning a 32-bit floating point value in two places.

Input:	Pointer to F32 dividend/result in DI. F32 divisor in PARM0 (4 bytes).
Output:	F32 result in memory location pointed to by DI. F32 result in PARM0 (4 bytes).
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	None.
Synopsis:	(DI),PARM0 ←(DI) / PARM0.
Stack Upon Entry:	Figure 9-3.

DIV_F32C*< size>*

Divides a 32-bit floating point value by another 32-bit floating point value, returning a 32-bit floating point value.

Input:	Pointer to F32 dividend/result in DI. F32 divisor in PARM0 (4 bytes).
Output:	F32 result in memory location pointed to by DI.
Registers Destroyed:	AX, BX, CX, DX.
Side Effects:	PARM0 is deallocated by this routine via RET 4.
Synopsis:	(DI) ←(DI) / PARM0.
Stack Upon Entry:	Figure 9-3.

Floating Point Comparison Routines

EQUAL_F64< *size*>

Compares two 64-bit floating point values, returning a 16-bit value of 0 if operand1 ≠ operand2, and 1 if operand1 = operand2.

Input:	F64 operand1 in PARM1 (8 bytes). F64 operand2 in PARM0 (8 bytes).
Output:	Boolean in AX where 0 = false, 1 = true.
Registers Destroyed:	CX, DX, DI.
Side Effects:	PARM0 and PARM1 are deallocated by this routine via RET 16.
Synopsis:	AX ← 1 if {PARM1 = PARM0} is true, 0 otherwise.
Stack Upon Entry:	Figure 9-2.

EQUAL_F32< *size*>

Compares two 32-bit floating point values, returning a 16-bit value of 0 if operand1 ≠ operand2, and 1 if operand1 = operand2.

Input:	F32 operand1 in register DXAX. F32 operand2 in register CXBX.
Output:	Boolean in AX where 0 = false, 1 = true.
Registers Destroyed:	BX, CX, DX, DI.
Side Effects:	None.
Synopsis:	AX ← 1 if {DXAX = CXBX} is true, 0 otherwise.

LESS_F64< *size*>

Compares two 64-bit floating point values, returning a 16-bit value of 0 if operand1 ≥ operand2, and 1 if operand1 < operand2.

Input:	F64 operand1 in PARM1 (8 bytes). F64 operand2 in PARM0 (8 bytes).
Output:	Boolean in AX where 0 = false, 1 = true.
Registers Destroyed:	CX, DX, DI.
Side Effects:	PARM0 and PARM1 are deallocated by this routine via RET 16.
Synopsis:	AX ← 1 if {PARM1 < PARM0} is true, 0 otherwise.
Stack Upon Entry:	Figure 9-2.

LESS_F32< *size*>

Compares two 32-bit floating point values, returning a 16-bit value of 0 if operand1 ≥ operand2, and 1 if operand1 < operand2.

Input:	F32 operand1 in register DXAX. F32 operand2 in register CXBX.
Output:	Boolean in AX where 0 = false, 1 = true.
Registers Destroyed:	BX, CX, DX, DI.
Side Effects:	None.
Synopsis:	AX ← 1 if {DXAX < CXBX} is true, 0 otherwise.

LESS_EQ_F64< *size*>

Compares two 64-bit floating point values, returning a 16-bit value of 0 if operand1 > operand2, and 1 if operand1 ≤ operand2.

Input:	F64 operand1 in PARM1 (8 bytes). F64 operand2 in PARM0 (8 bytes).
Output:	Boolean in AX where 0 = false, 1 = true.
Registers Destroyed:	CX, DX, DI.
Side Effects:	PARM0 and PARM1 are deallocated by this routine via RET 16.
Synopsis:	AX ← 1 if {PARM1 ≤ PARM0} is true, 0 otherwise.
Stack Upon Entry:	Figure 9-2.

LESS_EQ_F32< *size*>

Compares two 32-bit floating point values, returning a 16-bit value of 0 if operand1 > operand2, and 1 if operand1 ≤ operand2.

Input:	F32 operand1 in register DXAX. F32 operand2 in register CXBX.
Output:	Boolean in AX where 0 = false, 1 = true.
Registers Destroyed:	BX, CX, DX, DI.
Side Effects:	None.
Synopsis:	AX ←1 if {DXAX ≤ CXBX} is true, 0 otherwise.

Integer Multiplication Routines

MUL_I32A< *size*>

Multiplies two 32-bit long values (signed or unsigned), returning a 32-bit long value (signed or unsigned as appropriate).

Input:	I32 or UI32 multiplicand in register CXDI. I32 or UI32 multiplier in register AXBX.
Output:	I32 or UI32 result in register DXAX.
Registers Destroyed:	CX.
Side Effects:	None.
Synopsis:	DXAX $\leftarrow$CXDI $*$ AXBX

MUL_I32B< *size*>

Multiplies two 32-bit long values (signed or unsigned), returning a 32-bit long value (signed or unsigned as appropriate) in two places.

Input:	Pointer to I32 or UI32 multiplicand/result in DI. I32 or UI32 multiplier in register AXBX.
Output:	I32 or UI32 result in memory location pointed to by DI. I32 or UI32 result in register DXAX.
Registers Destroyed:	CX.
Side Effects:	None.
Synopsis:	(DI),DXAX $\leftarrow$(DI) * AXBX.

Integer Division Routines

DIV_UI32A*< size>*

Divides a 32-bit unsigned long value by another 32-bit unsigned long value, returning a 32-bit unsigned long value.

Input:	UI32 dividend in register CXDI. UI32 divisor in register AXBX.
Output:	UI32 result in register DXAX.
Registers Destroyed:	BX, DI.
Side Effects:	None.
Synopsis:	DXAX ←CXDI / AXBX.

DIV_UI32B*< size>*

Divides a 32-bit unsigned long value by another 32-bit unsigned long value, returning a 32-bit unsigned long value in two places.

Input:	Pointer to UI32 dividend/result in DI. UI32 divisor in register AXBX.
Output:	UI32 result in memory location pointed to by DI. UI32 result in register DXAX.
Registers Destroyed:	BX, CX.
Side Effects:	None.
Synopsis:	(DI),DXAX ←(DI) / AXBX.

DIV_I32A< *size*>

Divides a 32-bit signed long value by another 32-bit signed long value, returning a 32-bit signed long value.

Input:	I32 dividend in register CXDI. I32 divisor in register AXBX.
Output:	I32 result in register DXAX.
Registers Destroyed:	BX, CX, DI.
Side Effects:	None.
Synopsis:	DXAX ← CXDI / AXBX.

DIV_I32B< *size*>

Divides a 32-bit signed long value by another 32-bit signed long value, returning a 32-bit signed long value in two places.

Input:	Pointer to I32 dividend/result in DI. I32 divisor in register AXBX.
Output:	I32 result in memory location pointed to by DI. I32 result in register DXAX.
Registers Destroyed:	BX, CX.
Side Effects:	None.
Synopsis:	(DI),DXAX ← (DI) / AXBX.

Integer Modulo Routines

MOD_UI32A< *size*>

Divides a 32-bit unsigned long value by another 32-bit unsigned long value, returning a 32-bit unsigned long remainder.

Input:	UI32 dividend in register CXDI. UI32 divisor in register AXBX.
Output:	UI32 result in register DXAX.
Registers Destroyed:	CX, DI.
Side Effects:	None.
Synopsis:	DXAX ←CXDI **mod** AXBX.

MOD_UI32B< *size*>

Divides a 32-bit unsigned long value by another 32-bit unsigned long value, returning a 32-bit unsigned long remainder in two places.

Input:	Pointer to UI32 dividend/result in DI. UI32 divisor in register AXBX.
Output:	UI32 result in memory location pointed to by DI. UI32 result in register DXAX.
Registers Destroyed:	CX.
Side Effects:	None.
Synopsis:	(DI),DXAX ←(DI) **mod** AXBX.

MOD_I32A<*size*>

Divides a 32-bit signed long value by another 32-bit signed long value, returning a 32-bit signed long remainder.

Input:	I32 dividend in register CXDI. I32 divisor in register AXBX.
Output:	I32 result in register DXAX.
Registers Destroyed:	CX, DI.
Side Effects:	None.
Synopsis:	DXAX ← CXDI **mod** AXBX.

MOD_I32B<*size*>

Divides a 32-bit signed long value by another 32-bit signed long value, returning a 32-bit signed long remainder in two places.

Input:	Pointer to I32 dividend/result in DI. I32 divisor in register AXBX.
Output:	I32 result in memory location pointed to by DI. I32 result in register DXAX.
Registers Destroyed:	CX.
Side Effects:	None.
Synopsis:	(DI),DXAX ← (DI) **mod** AXBX.

Pointer and Range Fault Routines

FAULT_PTR*< size>*

Traps the appropriate error when a pointer is checked and found to be uninitialized or containing a NIL. A call to *_error_msg(fault_type, text_ptr, line_num)* is made with *fault_type* set to 0 for a NIL pointer and –1 for an unitialized pointer. *text_ptr* points to the filename and *line_num* is the line number.

Input:	Fault code number in register AX where: 0 = NIL pointer –1 = Uninitialized pointer *< pointer size>* -bit pointer at TOS to information block of the form: UI32 line number Filename (variable number of bytes) 0 (Filename terminator)
Output:	None.
Registers Destroyed:	N/A
Side Effects:	This routine may not be returned from.
Synopsis:	Call *_error_msg(fault_type, text_ptr, line_num)* and never return.

FAULT_UI32*< size>*

Traps the appropriate error when an unsigned long variable is checked and found to be outside of a predefined range. A call to *_error_msg(fault_type, text_ptr, line_num, value, limit)* is made with *fault_type* set to 1. *text_ptr* points to the filename, *line_num* is the line number, *value* is the bad index value, and *limit* is the index limit.

Input:	UI32 out of range index value in register DXAX *< pointer size>* -bit pointer at TOS to information block of the form: UI16 index limit UI32 line number Filename (variable number of bytes) 0 (Filename terminator)
Output:	None.
Registers Destroyed:	N/A
Side Effects:	This routine may not be returned from.
Synopsis:	Call *_error_msg(fault_type, text_ptr, line_num, value, limit)* and never return.

FAULT_UI16*<size>*

Traps the appropriate error when an unsigned integer variable is checked and found to be outside of a predefined range. A call to *_error_msg(fault_type, text_ptr, line_num, value, limit)* is made with *fault_type* set to 2. *text_ptr* points to the filename, *line_num* is the line number, *value* is the bad index value, and *limit* is the index limit.

Input:	UI16 out of range index value in register AX. *<pointer size>* -bit pointer at TOS to information block of the form: UI16 Index limit. UI32 Line number. Filename (variable number of bytes). 0 (Filename terminator).
Output:	None.
Registers Destroyed:	N/A
Side Effects:	This routine may not be returned from.
Synopsis:	Call *_error_msg(fault_type, text_ptr, line_num, value, limit)* and never return.

FAULT_UI8*< size>*

Traps the appropriate error when an unsigned char variable is checked and found to be outside of a predefined range. A call to *_error_msg(fault_type, text_ptr, line_num, value, limit)* is made with *fault_type* set to 3. *text_ptr* points to the filename, *line_num* is the line number, *value* is the bad index value, and *limit* is the index limit.

Input:	UI8 Out of range index value in register AL. *< pointer size>* -bit pointer at TOS to information block of the form: UI16 index limit UI32 Line number. Filename (variable number of bytes). 0 (Filename terminator)
Output:	None.
Registers Destroyed:	N/A
Side Effects:	This routine may not be returned from.
Synopsis:	Call *_error_msg(fault_type, text_ptr, line_num, value, limit)* and never return.

FAULT_I32*< size>*

Traps the appropriate error when a signed long variable is checked and found to be outside of a predefined range. A call to *_error_msg(fault_type, text_ptr, line_num, value, limit)* is made with *fault_type* set to 4. *text_ptr* points to the filename, *line_num* is the line number, *value* is the bad index value, and *limit* is the index limit.

Input:	I32 out of range index value in register DXAX. *< pointer size>* -bit pointer at TOS to information block of the form: UI16 Index limit. UI32 Line number. Filename (Variable number of bytes). 0 (Filename terminator)
Output:	None.
Registers Destroyed:	N/A
Side Effects:	This routine may not be returned from.
Synopsis:	Call *_error_msg(fault_type, text_ptr, line_num, value, limit)* and never return.

FAULT_I16*< size>*

Traps the appropriate error when a signed integer variable is checked and found to be outside of a predefined range. A call to *_error_msg(fault_type, text_ptr, line_num, value, limit)* is made with *fault_type* set to 5. *text_ptr* points to the filename, *line_num* is the line number, *value* is the bad index value, and *limit* is the index limit.

Input:	I16 out of range index value in register AX. *< pointer size>* -bit pointer at TOS to information block of the form: UI16 Index limit UI32 Line number. Filename (Variable number of bytes). 0 (Filename terminator)
Output:	None.
Registers Destroyed:	N/A
Side Effects:	This routine may not be returned from.
Synopsis:	Call *_error_msg(fault_type, text_ptr, line_num, value, limit)* and never return.

FAULT_I8*< size>*

Traps the appropriate error when a signed char variable is checked and found to be outside of a predefined range. A call to *_error_msg(fault_type, text_ptr, line_num, value, limit)* is made with *fault_type* set to 6. *text_ptr* points to the filename, *line_num* is the line number, *value* is the bad index value, and *limit* is the index limit.

Input:	I8 out of range index value in register AL. *< pointer size>* -bit pointer at TOS to information block of the form: UI16 Index limit. UI32 Line number. Filename (Variable number of bytes). 0 (Filename terminator)
Output:	None.
Registers Destroyed:	N/A
Side Effects:	This routine may not be returned from.
Synopsis:	Call *_error_msg(fault_type, text_ptr, line_num, value, limit)* and never return.

Stack Frame Figures

This section contains the figures that are referred to throughout this appendix.

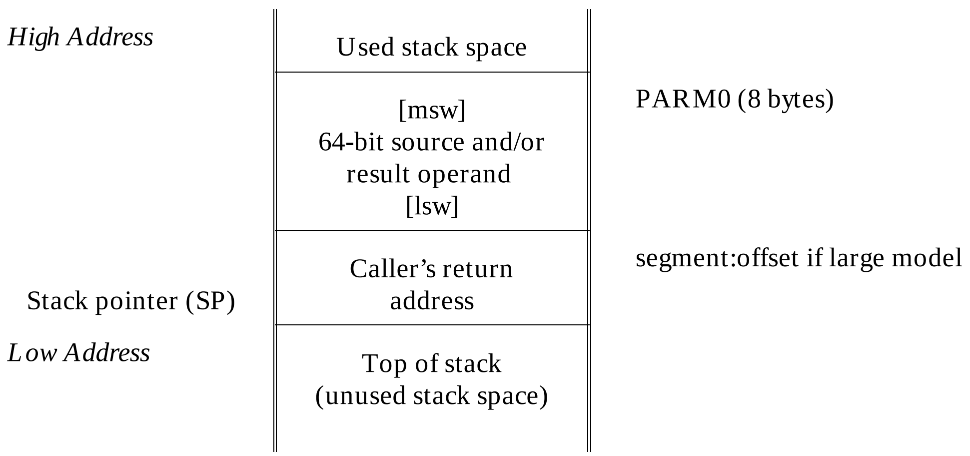

Figure 11-1. Stack Frame with Double Parameter

High Address	Used stack space	
	[msw] 64-bit left source and/or result operand [lsw]	PARM1 (8 bytes)
	[msw] 64-bit right source operand [lsw]	PARM0 (8 bytes)
Stack pointer (SP)	Caller's return address	segment:offset if large model
Low Address	Top of stack (unused stack space)	

Figure 11-2. Stack Frame with Two Double Parameters

High Address	Used stack space	
	[msw] 32-bit source and/or result operand [lsw]	PARM0 (4 bytes)
Stack pointer (SP)	Caller's return address	segment:offset if large model
Low Address	Top of stack (unused stack space)	

Figure 11-3. Stack Frame with Float or Long Parameter

High Address	Used stack space	
	[msw] 32-bit left source and/or result operand [lsw]	PARM1 (4 bytes)
	[msw] 32-bit right source operand [lsw]	PARM0 (4 bytes)
Stack pointer (SP)	Caller's return address	segment:offset if large model
Low Address	Top of stack (unused stack space)	

Figure 11-4. Stack Frame with 2 Float/Long Parameters

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.)

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
tanh	-1	R	x	0	0	x	R	1	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
	-∞	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
x	+ 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
	-∞	-∞	-∞	-∞	-∞	-∞	IOP NaN	y
	-N	-∞	R	x	x	R	+ ∞	y
	-0	-∞	y	-0	+ 0	y	+ ∞	y
x	+ 0	-∞	y	+ 0	+ 0	y	+ ∞	y
	+ N	-∞	R	x	x	R	+ ∞	y
	+ ∞	IOP NaN	+ ∞	+ ∞	+ ∞	+ ∞	+ ∞	y
	NaN	x	x	x	x	x	x	x

Table 12-5. "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 8086/186 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 8086/186 C compiler's comparable options are specified on the command line and affect the entire file.

All of the absolute (**.X**) files generated by the 8086/186 C Cross Compiler use a data bus width of 16 bits. If you used the directives "8088" or "80188" with C/64000, be aware that you must now specify the data bus width when programming PROMs. Thus instead of

```
program from file.X start 0 rom addr 0
```

you should use

```
program from file.X start 0 rom addr 0 system rom data width 0
```

If you do not specify the data width, the PROM will contain only alternate bytes from the 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 segment. In the new compiler, ORG functionality is accomplished via the **SEGMENT** pragma which is terminated by another **SEGMENT** 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 **SEGMENT** 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 8086/186 C compiler and is not implemented in this compiler. However, the **SEGMENT** pragma permits control over the segments 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 8086/186 C compiler.

8086-Specific C/64000 Options

ALIGN

By default, data and constants larger than one byte are aligned to a word boundary for efficient access. When the "byte align data" compiler option is used, data and constants are no longer necessarily aligned to a word boundary.

CS_EXTVARS, ES_EXTVARS, SS_EXTVARS

These are not implemented

DS_EXTVARS, FAR_EXTVARS

These are in effect supported. These are implemented via the SEGMENT pragma and the command line option that controls the memory model.

FAR_LIBRARIES, SHORT_LIBRARIES

These are not implemented. The FAR and NEAR aspects correspond to the large and small memory models respectively.

FAR_PROC, POINTER_SIZE

These can be implemented by choice of memory model and the "near calls" command line option in the large memory model.

INT

This is not implemented since the functionality can be accomplished by coding the "INT" with in-line assembly (see ASM and END_ASM pragmas).

INTERRUPT

This is implemented in the new C via the INTERRUPT pragma.

SEPARATE_CONST

Switch tables for "case" statements (jump vectors) are always placed in code segments. C constants (**const** declarations and strings) are put into their own CONST segment. Constants are ROMable in the large memory model, since the CONST segment can be placed adjacent to the PROG segment. In the small memory model, the CONST segment is more restricted in its placement in memory. See the section on "RAM and ROM Considerations" in the chapter "Embedded Systems Considerations" for more details.

Differences from HP 64818 Code

This section describes:

1 The differences between the HP 64818 and HP B1493 C compilers.

2 Ways to convert code written for the HP 64818 so that it will work with the B1493 C compiler.

Alignment

HP 64818	Word alignment is set by the $ALIGN ON$ option.
HP B1493	Word alignment is performed. Refer to the "Alignment Considerations" section in the "Internal Data Representations" chapter.

Integral promotions

HP 64818	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 B1493	The effect is the same as if integral promotions were always performed.

Float promotions

HP 64818	Promotion from a **float** to a **double** will be performed in an arithmetic operation unless $SHORT_ARITH ON$ is specified.
HP B1493	Promotion from a **float** to a **double** will not be performed unless one of the operands is a **double**.

Shift operations

HP 64818	Logical shift on all shift operations. Shift by a negative value will reverse the shift direction.
HP B1493	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 64818	Structures may be assigned, compared for equality, passed as parameters, or returned from functions.
HP B1493	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 64818	The first 15 characters in a symbol name are significant.
HP B1493	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 B1493.

Numeric constant formats

HP 64818	$EXTENSIONS ON$ permits use of HP 64000 format for defining binary, octal, decimal, and hexadecimal constants (e.g., 0FFH).
HP B1493	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 64818	Identical string constants or string constants that are a subset of another will be mapped into the same location to minimize space.
HP B1493	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 64818	INITHEAP, INCREASEHEAP, NEW, DISPOSE, MARK and RELEASE are provided for dynamic memory management.
HP B1493	*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 64818	ABS, SQRT, SIN, COS, ARCTAN, LN, and EXP are provided.
HP B1493	*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 64818	All signed and unsigned scalar values are extended to a 16-bit value and then pushed on the stack.
HP B1493	Same as HP 64818.

Passing a pointer

HP 64818	Pointers are pushed on the stack as 16 or 32 bit quantities as specified by the $POINTER_SIZE *n*$ option.
HP B1493	32-bit pointers are pushed on the stack for large memory model, 16-bit pointers for small memory model.

Passing a floating-point value

HP 64818	All floating point values are pushed on the stack as 64 bit double precision qualities, with the least significant bytes in lower memory addresses.
HP B1493	Same as HP 64818.

Passing a structure

HP 64818	Structures are pushed on the stack on word boundaries. The last word of the structure is passed first.
HP B1493	Same as HP 64818.

Passing an array

HP 64818	The address of the array is pushed on the stack.
HP B1493	Same as HP 64818.

Function return values

HP 64818	One byte results will be returned in register BL, two byte results in register BX. Return values greater than two bytes will be saved in the location pointed to by the result address pushed by the calling routine.
HP B1493	One byte results will be returned in register AL, two byte results in AX, three byte results in register pair DL, AX, and four byte results in register pair DX, AX. Return values greater than four bytes will be saved in the location pointed to by the result address pushed by the calling routine. This pointer may point to a static memory location, an automatic variable, or temporary space on the stack.

Removing parameters

HP 64818	If the $FIXED_PARAMETER$ option is OFF (default), the calling routine is responsible for removing parameters from the stack. If the option is ON, the parameters are removed by the called routine.

HP B1493	The calling routine is responsible for removing parameters from the stack.

Assembly Code Considerations

Stack frame management is different in the HP 64818 and HP B1493 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 B1449 assembler is used with the HP B1493 compiler.

Refer to the *8086/186 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 BP 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 B1493 compiler's calling conventions.

14

ASCII Character Set

Chapter 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

Stack Models

Diagrams of the five stack models used in the 8086/186 C Cross Compiler.

The stack models are:

- Stack Model for Small Memory Model
- Stack Model for Large Memory Model
- Stack Model for Medium Memory Model
- Stack Model for Compact Memory Model
- Near Stack Model for Large and Compact Memory Model (The near stack model applies when the "near calls" option is used.)
- Interrupt Stack Model for Large and Compact Memory Model
- Interrupt Stack Model for Small and Medium Memory Model

High Address	Used stack space	
	Reserved space for structure result	Absent if result is < = 4 bytes or if 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 < = 4 bytes. (Address size is 1 word.)
	Return address	(Address size is 1 word.)
Frame pointer (BP)	Old frame pointer (BP)	Absent if there are no parameters or locals. (Size is 1 word.)
	Last local ⇑ First local	Absent if function does not declare any local (automatic) variables. (Last declared local is first on stack.)
	Buffered register variable (SI)	Absent if function does not use register variables.
	Saved 8087 state	Present when -f option is used *and* floating point register variables are used. (Size is 94 bytes.)
Stack pointer (SP)	Temporaries ⇓	Stack changes as temporaries are saved and used in expressions.
Low Address	Top of stack	

Figure 15-2. Stack for Small Memory Model

High Address	Used stack space	
	Reserved space for structure result	Absent if result is < = 4 bytes or if result is returned through a variable.
	Last parameter ⇑ First parameter	Absent if no parameters are passed. (Last passed parameter is pushed first.)
	[segment] Result [offset] address	Absent if size returned is < = 4 bytes. (Address size is 2 words.)
	[segment] Return [offset] address	(Address size is 2 words.)
Frame pointer (BP)	Old frame pointer (BP)	Absent if there are no parameters or locals. (Size is 1 word.)
	Last local ⇑ First local	Absent if function does not declare any local (automatic) variables. (Last declared local is first on stack.)
	Buffered data segment (DS)	Absent if function does not access DS-relative static data.
	Buffered register variable (SI)	Absent if function does not use register variables.
	Saved 8087 state	Present when -f option is used *and* floating point register variables are used. (Size is 94 bytes.)
Stack pointer (SP)	Temporaries ⇓	Stack changes as temporaries are saved and used in expressions.
Low Address	Top of stack	

Figure 15-3. Stack for Large Memory Model

High Address	Used stack space	
	Reserved space for structure result	Absent if result is < = 4 bytes or if 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 < = 4 bytes. (Address size is 1 word.)
	[segment] Return [offset] address	(Address size is 2 word.)
Frame pointer (BP)	Old frame pointer (BP)	Absent if there are no parameters or locals. (Size is 1 word.)
	Last local ⇑ First local	Absent if function does not declare any local (automatic) variables. (Last declared local is first on stack.)
	Buffered register variable (SI)	Absent if function does not use register variables.
	Saved 8087 state	Present when -f option is used *and* floating point register variables are used. (Size is 94 bytes.)
Stack pointer (SP)	Temporaries ⇓	Stack changes as temporaries are saved and used in expressions.
Low Address	Top of stack	

Figure 15-4. Stack for Medium Memory Model

High Address	Used stack space	
	Reserved space for structure result	Absent if result is < = 4 bytes or if result is returned through a variable.
	Last parameter ⇑ First parameter	Absent if no parameters are passed. (Last passed parameter is pushed first.)
	[segment] Result [offset] address	Absent if size returned is < = 4 bytes. (Address size is 2 words.)
	Return address	(Address size is 1 words.)
Frame pointer (BP)	Old frame pointer (BP)	Absent if there are no parameters or locals. (Size is 1 word.)
	Last local ⇑ First local	Absent if function does not declare any local (automatic) variables. (Last declared local is first on stack.)
	Buffered data segment (DS)	Absent if function does not access DS-relative static data.
	Buffered register variable (SI)	Absent if function does not use register variables.
	Saved 8087 state	Present when -f option is used *and* floating point register variables are used. (Size is 94 bytes.)
Stack pointer (SP)	Temporaries ⇓	Stack changes as temporaries are saved and used in expressions.
Low Address	Top of stack	

Figure 15-5. Stack for Compact Memory Model

High Address	Used stack space	
	Reserved space for structure result	Absent if result is < = 4 bytes or if result is returned through a variable.
	Last parameter $\Uparrow$ First parameter	Absent if no parameters are passed. (Last passed parameter is pushed first.)
	[segment] Result [offset] address	Absent if size returned is < = 4 bytes. (Address size is 2 words.)
	Return address	(Address size is 1 word.)
Frame pointer (BP)	Old frame pointer (BP)	Absent if there are no parameters or locals, and size returned is < = 4 bytes. (Size is 1 word.)
	Last local $\Uparrow$ First local	Absent if function does not declare any local (automatic) variables. (Last declared local is first on stack.)
	Buffered data segment (DS)	Absent if function does not access DS-relative static data.
	Buffered register variable (SI)	Absent if function does not use register variables.
	Saved 8087 state	Present when -f option is used *and* floating point register variables are used. (Size is 94 bytes.)
Stack pointer (SP)	Temporaries $\Downarrow$	Stack changes as temporaries are saved and used in expressions.
Low Address	Top of stack	

Figure 15-6. NEAR Stack for Large and Compact Model

High Address	Used stack space	
	[segment] Interrupt return [offset] address	
	Processor flags	
	Old AX	
	Old CX	
	Old DX	
	Old BX	
	Old DI	
	Old ES	
Frame pointer (BP)	Old frame pointer (BP)	Absent if there are no locals. (Size is 1 word.)
	Last local $\Uparrow$ First local	Absent if function does not declare any local (automatic) variables. (Last declared local is first on stack.)
	Buffered data segment (DS)	Absent if function does not access DS0-relative static data.
	Buffered register variable (SI)	Absent if function does not use register variables.
	Saved 8087 state	Present if **-f** is used (94 bytes).
Stack pointer (SP)	Temporaries $\Downarrow$	Stack changes as temporaries are saved and used in expressions.
Low Address	Top of stack	

Figure 15-7. Interrupt Stack for Large & Compact Model

High Address	Used stack space	
	[segment] Interrupt return [offset] address	
	Processor flags	
	Old AX	
	Old CX	
	Old DX	
	Old BX	
	Old DI	
Frame pointer (BP)	Old frame pointer (BP)	Absent if there are no locals. (Size is 1 word.)
	Last local ⇑ First local	Absent if function does not declare any local (automatic) variables. (Last declared local is first on stack.)
	Buffered register variable (SI)	Absent if function does not use register variables.
	Saved 8087 state	Present if **-f** is used (94 bytes).
Stack pointer (SP)	Temporaries ⇓	Stack changes as temporaries are saved and used in expressions.
Low Address	Top of stack	

Figure 15-8. Interrupt Stack for Small & Compact Model

16

About this Version

How this version of the compiler differs from previous versions.

Version 4.01

New memory models

The compact and medium memory models are supported. The "-m compact" option tells the compiler to generate code for the compact memory model. The "-m medium" option tells the compiler to generate code for the medium memory model.

Control of NOPs

The -Wc,Hx option allows you to specify the number of NOPs between functions. The default number is 1.

C++ style comments

C++ style comments are now accepted by cpp8086.

Enhanced -M option

The compiler warns when a function is used without a previously declared prototype if the -M command-line option is used.

New usage message

cc086 prints a usage message if no options are used on the command line.

Version 4.00

New product number

The product number has been changed to B1493 for all hosts.

The old product number was 64904 (for HP 300/400 hosts), and B1427 (for Apollo hosts—no longer supported).

New command-line options

The **-Wo,m** option tells the optimizer to avoid certain optimizations.

The **-K** option enforces strict segment information consistency.

New default environments

All of the default environments supplied with the compiler are now HP 64700-series emulators.

Renamed run-time library routines

Some run-time library routines have been renamed in anticipation of the addition of medium and compact memory models. Routines which will be supported by both the large and medium models now have a _LM suffix. Routines which will be supported by both the small and compact models have a _SC suffix.

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.

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 "> = ".

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/hp6476*x*/iolinkcom.k) show how this can be done.

Improved performance

The compile speed has been significantly improved. @S2 = 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 8086/186 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 B1449 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.

New segment names

All compiler-generated code is now placed in segments with the class name "CODE." Thus there are now segment names in the **.k** files such as "lib/CODE" and "libm/CODE" in place of "lib" and "libm."

This change affects the *iolinkcom.k*, *linkcom.k*, *fiolinkcom.k*, and *flinkcom.k* files in */usr/hp64000/env/hp6476x/*large, */usr/hp64000/env/hp6476x/*small, and the corresponding directories for the other supported emulators.

Library constants are no longer placed in the same segment as the code. The constants for "libc" and "libm" are now placed in segments "libcconst" and "libmconst," respectively.

If you will be using the **.k** that are shipped with the compiler, these changes will not affect you. If, however, you have modified **.k** files for a previous version of the compiler, you will need to add the new section names.

17

On-line Manual Pages

Printed copies of the on-line documentation.

cc8086 (1)

NAME

cc8086 - C cross compiler for Intel 8086 microprocessor

SYNOPSIS

/usr/hp64000/bin/cc8086 [options] files
/usr/hp64000/bin/cc80186 [options] files

DESCRIPTION

The *cc8086* program is a C cross-compiler which generates object code for the Intel 8086 microprocessor. *Cc80186* generates object code for the Intel 80186 microprocessor. They accept several types of arguments:

Arguments whose names end with **.c** are taken to be C source programs. They are compiled and each object program is left on the file whose name is that of the source with **.o** substituted for **.c**. The **.o** file is deleted only if a single C program is compiled and linked all in one step.

In the same way, arguments whose names end with **.s** are taken to be assembly source programs and are assembled, producing a **.o** file.

Arguments whose names end with **.i** are taken to be C source programs which have already been preprocessed (see **-P**). They are compiled without invoking *cpp8086*(1) and each object program is left on the file whose name is that of the source with **.o** substituted for **.i**.

Arguments whose names end with **.o** are taken to be relocatable object files which are to be included in the link operation.

Arguments can be passed to the compiler through the **CC8086OPTS** environment variable as well as on the command line. The compiler picks up the value of **CC8086OPTS** and places its contents before any arguments on the command line. For example (in *sh*(1) notation):

```
CC8086OPTS= -v
export CC8086OPTS
cc8086 -L prog.c
```

is equivalent to:

```
cc8086 -v -L prog.c
```

The compiler also checks the environment variable **HP64000**. If it has been set and exported, it is used as the directory path (in place of the default */usr/hp64000*) for executables (e.g., */lib/cpp8086*), libraries specified by using **-l** (e.g. */lib/8086/large/libm.a*), include files (e.g. */include/8086/stdio.h*), and the default linker command file (*/env/hp6476x/large/iolinkcom.k*).

The following options are recognized by *cc8086*:

-b

Cause the compiler to use the Branch Validator preprocessor, which inserts additional code for branch counting. See *bbacpp8086*(1).

-c

Suppress the link edit phase of the compilation and force an object (**.o**) file to be produced even if only one program is compiled. Produces a **.o** file for each **.c** file.

-C

Do not strip C-style comments in the preprocessor except those found on preprocessor directive lines. See *cpp8086*(1).

-d

Separate data output into initialized (non-constant data explicitly initialized with a C initializer) and uninitialized (non-constant data implicitly initialized to zero, in the absence of **-u**, at load time). The default output segments are *udata* and *idata*. See also **# pragma SEGMENT**.

-D *name= def*
-D *name*

Define *name* to the preprocessor. See *cpp8086*(1).

-e

Turn off code generation allowing fast syntactic and semantic error checking of the source program. This option overrides the **-L**, **-c**, and **-S** options.

-E

Run only *cpp8086*(1) on the named C programs and send the result to the standard output.

-f

Generate 8087 (floating point coprocessor) code for floating point operations. This option causes code to be generated in-line for operations which might otherwise be performed with run-time library calls. It also causes linker command file *fiolinkcom.k* (or *flinkcom.k*, if **-N** used) to be used. These linker command files use the 8087 run-time (*lib87.a*) and math (*libm87.a*) library files.

-g

Generate additional (but less optimal) code which performs run-time error checking. Note that it is not necessary to use **-g** to get complete symbolic debugging.

The two types of run-time checks made are:

1) Dereferences of all NIL pointers and uninitialized automatic pointers are detected and reported. This requires the initialization of automatic pointers at run-time with a value (-1) indicating that they are uninitialized. Note that initialization of statics to the uninitialized pointer value is not possible, as statics default to zero.

2) Array references outside declaration index bounds are detected and reported.

This option overrides the **-O** or **-s** option.

-h

Cause generation of an HP 64000 format assembler symbol file, linker symbol file, and absolute file for debugging purposes. The assembler symbol file is named (source basename)**.A**, the linker symbol file (output file name)**.L**, and the absolute file (output file name)**.X**. The symbol file names can be changed with the **-H** assembler or linker option passed via the **-W** option.

-I *dir*

Change the search algorithm used by the preprocessor for finding include files. See *cpp8086*(1).

-k *linkcomfile*

Cause the named *linkcomfile* to be used by the linker rather than the default */usr/hp64000/env/hp6476x/large/iolinkcom.k* (see also **-f**, **-N** and **-r**).

Note that if the environment variable **HP64000** is set and exported, the */usr/hp64000* part of the path for the default file becomes *$HP64000*.

See *ld86*(1) for details about the format of linker command files.

The **-k** option overrides any linker command file implications of the **-f**,

-N, **-p**, or **-r** options.

-K

Cause the compiler to strictly enforce section information for variables. By default, the compiler does not require that the section information between a symbol declaration and definition match exactly. This option forces the information to be identical.

Section information for variables and functions are communicated to the compiler via the **SEGMENT** pragma. With this information the compiler can address different sections of code and data with different address modes. If a different section is named for a "extern" reference than the actual variable declaration, then undesired addressing modes could be used. This could lead to a defect in code generation that is very difficult to locate. Usage of the **-K** option will cause this type of coding error to be found at compile time. Its use is highly recommended.

See also the discussion about **# pragma SEGMENT**.

-l*x*

Cause the linker to search the library **/usr/hp64000/lib/8086/large/lib***x***.a** (or **/usr/hp64000/lib/8086/small/lib***x***.a** if **-m small** is used, or **/usr/hp64000/lib/8086/medium/lib***x***.a** if **-m medium** is used, or **/usr/hp64000/lib/8086/compact/lib***x***.a** if **-m compact** is used,). Use **-l ""** to load **lib.a**. If the environment variable **HP64000** is set and exported, the */usr/hp64000* part of the path becomes *$HP64000*. Note that **-l** options must appear *after* any files which reference library routines, typically at the *end* of the command line. You do not need to use the **-l** option if the library is loaded by the linker command file.

-L[**i**][**x**]

Cause the compiler to generate a listing file (suffixed with **.O**) for each C source compiled. This listing contains C source intermixed with generated assembly code.

If the **-S** (do not assemble) option is present, the intermixed assembly is just as it appears in the **.s** file; otherwise, the intermixed assembly is taken from the assembler's listing file with program counters and object code.

If the **-i** option is present, include files are expanded and included in the listing.

If the **-x** option is present, a symbol cross-reference table is appended to the compiler listing and also to any assembler or linker listings.

If the assembler is invoked for any **.s** files on the command line, an assembler listing is produced in the corresponding listing file suffixed **.O** and **-x** invokes the assembler's cross-reference.

If the linker is invoked, a linker listing is produced in a listing file named *outfile***.O** (default *a.out.O*, see **-o**) and **-x** invokes the linker's cross-reference.

Options which prevent compilation (**-e**, **-E**, and **-P**) prevent the generation of listings.

-m *memoryModel*

Cause the compiler to generate code for the selected memory model. If this option is not present, the large memory model is assumed. *memoryModel* may be either:

large large memory model (default)
small small memory model
medium for the medium memory model
compact for the compact memory model

The small memory model has two segments which never change. One is a code segment (CS does not change). The other is a combined stack and DS-relative static data segment (DS, SS, and ES are identical and do not change). In this model all pointers are 16 bits.

The large memory model may have one or more code segments (CS may change); one independent stack segment (SS does not change); zero or one DS-relative static data segment for each C function (DS may change); and zero, one, or more ES-relative static data segments (ES may change). In this model all pointers are 32 bits. Functions are considered to be "FAR" and are called as such except when a static function is encountered and the **-n** option is in effect.

The medium memory model may have one or more code segments (the CS register may change) and one data segment (the DS, SS, and ES registers are identical and do not change). The function pointer size is 32 bits, and the data pointer size is 16 bits.

The compact memory model has one fixed code segment (the CS register does not change) and one or more data segments (the DS, SS, and ES registers are not identical and may change). The function pointer size is 16 bits, and the data pointer size is 32 bits.

-M

Cause the compiler to generate more warning messages for possible errors in the C source than are generated by default.

-n

Cause all static functions in large memory model to be called "NEAR". This option should be used only when the user can guarantee that all static functions within the source file(s) being compiled are called from the same segment and that no pointer arithmetic is being performed to generate the call. This option is ignored in the presence of **-m small**.

-N

Cause the compiler to link using the *linkcom.k* (*flinkcom.k*, if **-f** used) linker command file rather than the *iolinkcom.k* (*fiolinkcom.k*, if **-f** used) command file. The [*f*]*linkcom.k* command file loads the *crt1.o* program setup routine which does not open *stdin*, *stdout*, or *stderr*. **-N** is overridden by the **-k** option, but works in conjunction with the **-r** option.

-o *outfile*

Name the output file from the linker *outfile***.x** (or *outfile***.X** and *outfile***.L** if the **-h** option is specified). *Outfile* is **a.out** by default.

-O[**G**][**T**]

Generate locally optimal code and invoke an assembly code optimizer. Code is optimized for space (even, possibly, at the expense of time) unless **-T** is also specified.

If the **-T** option is present, code is optimized for time whenever time and space optimizations conflict. If the **-G** option is present, additional code is generated (as it is when **-O** is not used) to make the program easier to debug using an HP emulator or debugger. This includes:

> 1) Generation of no-operation (**NOP**) instructions preceding all labels. This provides unique addresses for all labels. Note that the peephole optimizer will remove any of these no-operation instructions that it considers to be dead code (following an unconditional branch).
>
> 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 not to be necessary.

The **-O** option is overridden by the **-g** option.

-P

Run only *cpp8086*(1) on the named C programs and leave the result on corresponding files suffixed **.i**.

-Q

Cause the compiler to byte align data in memory, rather than the default word alignment. Data of types short, int, long, pointer, float, double, struct, and union will be aligned on byte rather than word boundaries. Data of types struct and union will not be padded. Note that the size of structures and unions as well as the offsets of their members are affected by this option. Therefore, modules which define structures or unions and those which reference them or their members must both be compiled with the same alignment. For the sake of safety, it is recommended that **all** sources linked together be compiled with the same alignment. *libc.a*, *libm.a*, and the run-time (*lib.a*) and environment libraries (*env.a*) are compatible with modules compiled under either alignment.

-r *dir*

Cause the compiler to use default linker command files *iolinkcom.k*, *linkcom.k*, *fiolinkcom.k*, or *flinkcom.k* (see **-N** and **-f**) in run-time environment directory **/usr/hp64000/env/***dir***/large** (or **/usr/hp64000/env/***dir***/[mem_model]** if **-m [mem_model]** present) rather than in the default */usr/hp64000/env/hp6476x/large*. For the Intel family of the HP 64700 series of emulators, the run-time environment is *hp6476x*. If the environment variable **HP64000** is set and exported, the */usr/hp64000* part of the path for the above environments becomes *$HP64000*. The **-r** option is overridden by the **-k** option, but works in conjunction with the **-N** and **-m** options.

-s

Cause the output of the compiler, assembler, and linker to be stripped of symbol table information. The use of this option will prevent the use of symbols for analysis/debug purposes in any consumers of the executable. This option is overridden by the **-g** option and, for file and line information, by the **-L** option.

-S

Compile the named C programs and leave the assembly language output on corresponding files suffixed **.s**. This option prevents invocation of the assembler.

-t *c,name*

Substitute or insert subprocess *c* with *name* where *c* is one or more of a set of identifiers indicating the subprocess(es). This option works in one of two modes:

1) If *c* is a single identifier, *name* represents the full path name of the new subprocess. For example: cc8086 -tp,/mydir/cpp source.c

2) If *c* is a set of identifiers, *name* represents a prefix to which the standard suffixes are concatenated to construct the full path names of the new subprocesses. For example: cc8086 -tpc2L,/mydir/ -tal,/mydir2/ source.c

c can be one or more of the following identifiers:

p	preprocessor (standard suffix is cpp8086)
c	compiler body (standard suffix is ccom8086L or ccom8086S)
0	same as c
o	optimizer (standard suffix is opt8086)
2	same as o
m	macro preprocessor for assembler (standard suffix is ap86)
a	assembler (standard suffix is as86)
L	lister (standard suffix is clst8086)
l	linker (standard suffix is ld86)

Note also in this context that the standard processes invoked are named:

/usr/hp64000/lib/cpp8086
/usr/hp64000/lib/ccom8086L (or ccom8086S, ccom8086C, or ccom8086M)
/usr/hp64000/lib/opt8086
/usr/hp64000/bin/ap86
/usr/hp64000/bin/as86
/usr/hp64000/lib/clst8086
/usr/hp64000/bin/ld86

If the environment variable **HP64000** is set and exported, the */usr/hp64000* part of the path for the above files becomes *$HP64000*.

-u

Cause the compiler to consider all non-constant static data as uninitialized and to issue a warning whenever an initializer is placed on such static data. This option is useful for embedded environments where no load-time initialization is possible (as opposed to environments, such as emulation or simulation, where load-time initialization of static data is possible only when the user loads the memory).

-U *name*

Remove any initial definition of *name*, where *name* is a reserved symbol that is predefined by the preprocessor or a symbol defined by a **-D** option regardless of the order of the options. Normally, *cc*(1) predefines symbols that reflect simultaneously the host and execution environment. Since *cc8086* is a cross development tool, it predefines one symbol indicating the cross environment and one target processor symbol. Additionally, a symbol is predefined to indicate which memory model is being used. The reserved symbols are:

cross environment: __hp64000
target processor: __i8086
memory model: __LARGE_MODEL or __SMALL_MODEL or __MEDIUM_MODEL or __COMPACT_MODEL

-v

Enable verbose mode, producing a step-by-step description of the compilation process on *stderr*.

-w

Suppress warning messages.

-W *c,arg1[,arg2,...,argN]*

Cause *arg1* through *argN* to be passed as parameters to subprocess *c* of the compilation process. The *args* are of the form -*argoption[,argvalue]*, where *argoption* is the name of an option to be passed to the subprocess and *argvalue* is an argument to *argoption*. The valid values for *c* are those listed under the **-t** option. For example, to invoke the **-t** option of *clst8086*(1): cc8086 -L -WL,-t source.c

Note that options other than the above are not recognized and cause a warning message to be written to *stderr*.

The following options to *ccom8086* are accessible via the **-W** option described above:

-C *segname*

Change the default segment name for constant output (see **CONST** under **# pragma SEGMENT** below) from the default *const* to the argument, *segname*.

-D *segname*
-D *segname1,segname2*

If one argument is given, change the default segment name for data output (see **DATA** under **# pragma SEGMENT**) from the default *data* to the argument, *segname*.

If two arguments are given, change the default segment name for uninitialized data output (see **UDATA** under **# pragma SEGMENT**) from the default *udata* to the first argument, *segname1*, and change the default segment name for initialized data output (see **IDATA** under **# pragma SEGMENT**) from the default *idata* to the second argument, *segname2*.

-F

Turn off the compiler's automatic creation of register variables for addresses of statics and frequently used variables.

-l *prefix*

Alter the compiler's algorithm for creating assembly language labels from C symbols. Rather than using an underscore at the beginning of such labels, the compiler will use *prefix*. Since the default *prefix* is '_', it is used as a special case for specifying that no prefix be used via -Wc,-l_. This option should be **used with great care** as it may generate assembly-time and/or link-time errors due to conflicts between compiler-generated assembly language labels and other assembly symbols. See also **# pragma ALIAS**.

-N *modulename*

Cause the compiler to use *modulename* for the argument to the **NAME** directive in the assembly source produced rather than the default modulename which is the C source file basename (see *basename*(1)).

-P *segname*

Change the default segment name for program output (see **PROG** under **# pragma SEGMENT**) from the default prog_*basename* to the argument, *segname*.

In addition to standard C in the C source files, *cc8086* accepts and ignores all pragmas except the following:

 # pragma ALIAS *Csymbolname Assemblysymbolname*
 # pragma ALIAS *Csymbolname* """*Assemblysymbolname*"""

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

 # pragma ASM
 # pragma END_ASM

These pragmas are used to bracket sections of assembly code which are inserted into the assembly code generated by the compiler. The assembly code optimizer assumes that working registers AX, BX, CX, DX, DI, processor status word (PSW), ST(0) and ST(1) when using **-f**, and in large memory model ES are destroyed in embedded assembly code sections; therefore, they may be freely used. The register variable (SI), the frame pointer (BP), the stack pointer (SP), the segment registers (CS, DS, SS, and, for small memory model, ES), and the floating point register variables (ST(2) through ST(6)) when using **-f** are not buffered prior to embedded assembly language sections. So, inadvertently writing over one of these registers should be avoided. Also, when using the **-f** option the 8087 stack pointer must not be left in an altered state and ST(7) must be "free". Embedded assembly code may reference C variables. The compiler incorporates the C name of variables and functions into their corresponding assembly code symbols facilitating the referencing of C variables from embedded assembly code. In particular: externs, globals, statics, and functions have an underscore (_) prepended to their C name and, for each parameter and automatic, an assembly-time constant is created by prepending S_ to its C name. This constant is a frame-pointer-relative offset used to access the parameter or automatic value. Because of scoping requirements and a 30 character significance limit on assembly names, C names longer than 29 characters and those which appear in nested scopes may have an additional ASCII number prepended to make them unique. See the COMPILER GENERATED ASSEMBLY CODE chapter in the manual for a complete discussion and examples of assembly symbol name generation. The **ASM** pragma may be used any place in a C source (i.e. inside or outside of a C function).

 # pragma DS *segmentName*

This pragma is only valid for large memory model; it is ignored in the presence of **-m small**. This pragma specifies that all subsequent functions should arrange to access any data in segment *segmentName* (rather than the default of the current **DATA** or **UDATA** segment name, see **# pragma SEGMENT**) using DS-relative addressing. If subsequent functions access any static data in segment *segmentName*, their preambles load DS with *segmentName* and use it in accesses. The effect of this is that once a **DS** pragma is used, the DS-relative segment name is fixed until another **DS** pragma is encountered.

 # pragma FUNCTION_ENTRY "*C_String***"**
 # pragma FUNCTION_EXIT "*C_String***"**
 # pragma FUNCTION_RETURN "*C_String***"**

These pragmas also allow you to insert assembly code into the generated assembly code. They differ from the **ASM** pragma in several ways:

They are not required to be paired and may be used independently or together. They may only appear outside of a function body. They affect only a single function and must precede that function in the C source. They do not bracket the embedded assembly. Instead, the assembly is contained in the "*C_String*" argument. This argument is a C character string. It must contain whitespace and newlines so that when the compiler outputs the string to the generated source, it will conform to the format and syntax required by the assembler. **# pragma FUNCTION_ENTRY** will place the embedded assembly in a function's entry code. The embedded code appears immediately after the label generated from the function name in the C source and will precede the code generated for function entry. **# pragma FUNCTION_EXIT** will place the embedded assembly in a function's exit code. The embedded code appears after the code generated for the function exit and precedes the function return label. **# pragma FUNCTION_RETURN** will place the embedded assembly in a function's exit code. The embedded code appears after the return label. These pragmas give you the flexibility to modify the function entry and exit code. An example is using FUNCTION_RETURN to force an interrupt service routine to trap back to the operating system instead of simply returning to the point of interrupt. (See also **# pragma INTERRUPT**.) The information found under

 # pragma ASM about accessing C symbols and about register buffering holds true for these pragmas as well.

 # 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 and restored prior to function exit (in addition to the register variable which ordinarily is buffered), no parameter passing or returned result is allowed, and a return from interrupt is generated at the return point. In the presence of the **-f** option, the 8087's complete internal state is saved. 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.

 # pragma SEGMENT [**PROG=** *pname*] [**DATA=** *dname*] [**CONST=** *cname*]

 # pragma SEGMENT [**PROG=** *addr*] [**DATA=** *addr*] [**CONST=** *addr*]

 # pragma SEGMENT [**PROG=** *pname*] [**UDATA=** *udname*] [**IDATA=** *idname*]
[**CONST=** *cname*]

 # pragma SEGMENT [**PROG=** *addr*] [**UDATA=** *addr*] [**IDATA=** *addr*]
[**CONST=** *addr*]

 # pragma SEGMENT UNDO
 # pragma SEGMENT [**PROG=** *pname*] [**DATA=** *dname*] [**CONST=** *cname*]

 # pragma SEGMENT [**PROG=** *addr*] [**DATA=** *addr*]

[**CONST=** *addr*]

 # pragma SEGMENT [**PROG=** *pname*] [**UDATA=** *idname*]
[**IDATA=** *udname*] [**CONST=** *cname*]

 # pragma SEGMENT [**PROG=** *addr*] [**UDATA=** *addr*]
[**IDATA=** *addr*] [**CONST=** *addr*]

 # pragma SEGMENT UNDO
This pragma is valid for large, medium, and compact memory model; it is ignored in the presence of **-m small**. The first form of this pragma causes the program, static data, and static constant information to be placed in segments named *pname*, *dname*, and *cname* respectively until the next **SEGMENT** pragma is encountered. This segment information is used for specifying the location of symbols to the linker. The linker expects to find external data in the segment whose name is active when the external declaration is made. In the second form, 20-bit physical addresses, whose syntax is the same as for C

constants, are given in place of the segment names causing the subsequent information to be ORG'd starting at the given address. The segment name associated with an ORG'd segment is of the form **org***hexaddress* where *hexaddress* is the physical address where the segment is located. For example, org00012345H is located at 0x12345. 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 **-d** option 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 segment. Non-constant static data items not explicitly initialized by means of a C initializer go into the **UDATA** named segment. Always use **DATA** (as opposed to **UDATA** or **IDATA**) to locate an external declaration in a segment. Note that changing **DATA** also changes both **UDATA** and **IDATA**. 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 **SEGMENT** pragma. If the target segment is not specified for one of the information types, then it remains unchanged. The last form, **# pragma SEGMENT UNDO**, "undoes" the effect of the immediately preceding **SEGMENT** directive. That is, it restores the name (or address) of any segment renamed (or ORG'd) in the last directive. This form is useful at the end of **# include** files to restore the segment environment which existed prior to the **# include** file. (Include files should contain **SEGMENT** directives to define the segments that externs are in.) Default (without **-d**):
PROG= prog_*basename* **DATA=** data **CONST=** const Default (with **-d**):
PROG= prog_*basename* **UDATA=** udata **IDATA=** idata **CONST=** const
basename is the C source file base name (see *basename*(1)) with all characters not legal for a segment name changed to underscore (_).

Note that pragmas other than the above are not recognized and cause a warning message to be written to *stderr*.

FILES

file.c	C source file
file.s	assembly source file
file.o	object file
file.a	library (archive) file
/usr/hp64000/lib/cpp8086	preprocessor
/usr/hp64000/lib/ccom8086L	compiler for large memory model

/usr/hp64000/lib/ccom8086S	compiler for small memory model
/usr/hp64000/lib/ccom8086C	compiler for compact memory model
/usr/hp64000/lib/ccom8086M	compiler for medium memory model
/usr/hp64000/lib/opt8086	optimizer
/usr/hp64000/bin/ap86	macro preprocessor for assembler
/usr/hp64000/bin/as86	assembler
/usr/hp64000/lib/clst8086	lister (C listing generator)
/usr/hp64000/bin/ld86	linker
/usr/hp64000/include/8086	standard directory for **# include** files

Note that, when environment variable **HP64000** is set and exported, it replaces "/usr/hp64000" in all of the above file names.

/usr/hp64000/lib/8086/large/lib.a	run-time library
/usr/hp64000/lib/8086/large/libc.a	standard C support library
/usr/hp64000/lib/8086/large/libm.a	auxiliary math C support library
/usr/hp64000/lib/8086/large/lib87.a	run-time library using 8087
/usr/hp64000/lib/8086/large/libm87.a	auxiliary math C support library using 8087
/usr/hp64000/env/hp6476x/large/env.a	execution environment dependent library
/usr/hp64000/env/hp6476x/large/iolinkcom.k	default linker command file
/usr/hp64000/env/hp6476x/large/linkcom.k	linker command file when **-N** (no I/O) used
/usr/hp64000/env/hp6476x/large/fiolinkcom.k	linker command file when **-f** (8087 code) used
/usr/hp64000/env/hp6476x/large/flinkcom.k	linker command file when **-N** (no I/O) and **-f** (8087 code) used

/usr/hp64000/env/hp6476x/large/crt0.o
default program setup routine

/usr/hp64000/env/hp6476x/large/crt1.o
program setup routine with no I/O initialization

/usr/hp64000/env/hp6476x/large/div_by_0.o
integer divide by zero interrupt routine

/usr/hp64000/env/hp6476x/large/vector8087.o 8087
exceptions interrupt routine

Note that when the **-m small** (small memory model), **-m medium** (medium memory model), or **-m compact** (compact memory model) option is used the *large* in the above paths is changed to *small*, *medium*, or *compact*.

/usr/hp64000/env/hp6476x/ioconfig.EA
emulation configuration file corresponding to
iolinkcom.k and *fiolinkcom.k* if present

/usr/hp64000/env/hp6476x/config.EA
emulation configuration file corresponding to
linkcom.k and *flinkcom.k* if present

/usr/hp64000/env/hp6476x/fioconfig.EA
emulation configuration file corresponding to *fiolinkcom.k*

/usr/hp64000/env/hp6476x/fconfig.EA
emulation configuration file corresponding to *flinkcom.k*

/usr/hp64000/env/hp6476x/src
directory containing sources for environment
dependent routines and emulation monitor

See the **-r** option for easy access.

AUTHOR

The *cc8086* program was developed by the Hewlett-Packard Company.

SEE ALSO

ap86(1), ar86(1), as86(1), bbacpp8086(1), clst8086(1), cpp8086(1), ld86(1)

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, 1988
HP 8086/186 C Cross Compiler User's Guide, Hewlett-Packard, 1995

DIAGNOSTICS

cc8086 returns zero if no errors are detected during the compilation process, otherwise it returns non-zero.

The diagnostics produced by *cc8086* are intended to be self-explanatory. Occasional messages may be written to *stderr* by the assembler. Error messages produced by the compiler are always written to *stderr* and consist of the original C source line on which the error was detected followed by a line containing a pointer to the token at which the error was detected and an explanatory message. Note that, for syntax errors, the token indicated will often be the token *following* the error.

cpp8086(1)

NAME

cpp8086 **-** C cross language preprocessor for Intel 8086 microprocessor

SYNOPSIS

/usr/hp64000/lib/cpp8086 [options] ifile [ofile]

DESCRIPTION

The *cpp8086* command is the ANSI/ISO standard (9899-1990) C language preprocessor which is invoked as the first pass of any C compilation using the *cc8086*(1) command. Its purpose is to process **include** files, conditional compilation instructions, and macros. Thus the output of *cpp8086* is designed to be in a form acceptable as input to the next pass of the C compiler. The preferred way to invoke *cpp8086* is through the *cc8086*(1) command, since the functionality of *cpp8086* may someday be moved elsewhere. Therefore, the direct invocation of *cpp8086* is not recommended. See *m4*(1) for a general macro processor.

The *cpp8086* command optionally accepts one or two file names as arguments. The arguments *ifile* and *ofile* are respectively the input and output for the preprocessor. If *ofile* is not supplied it defaults to standard output.

The following options to *cpp8086* are recognized:

-P

Preprocess the input without producing the line control information used by the next pass of the C compiler.

-C

By default, *cpp8086* strips C-style comments. If the **-C** option is specified, all comments (except those found on *cpp8086* directive lines) are passed along.

-U *name*

Remove any initial definition of *name*, where *name* is a symbol defined by a **-D** option regardless of the order of the options.

-D *name*
-D *name*= *def*

Define *name* as if by a **# define** directive. If no = *def* is given, *name* is defined as 1. The **-D** option has lower precedence than the **-U** option. That is, if the same name is used in both a **-U** option and a **-D** option, the name is undefined regardless of the order of the options.

-I dir Add *dir* to the directory search list for **# include** files whose names do not begin with /. Thus, **# include** files whose names are enclosed in "\| "are searched for first in the directory of the file containing the **# include** line, then in directories named in **-I** options in left-to-right order. For **# include** files whose names are enclosed in < > , the directory of the file containing the **# include** line is not searched. However, all directories specified with **-I** options will still be searched.

To access the standard header files shipped with the C compiler, add the directory */usr/hp64000/include/8086* to the search list via this option.

-g

Causes *cpp8086* to generate file date and column position information. File date information is appended to the line and file synchronization information which is normally generated indicating the last modified date of the source and include files. Column position synchronization information is provided whenever macro substitution takes place. The line 'this is a line', where 'is' is a macro defined to be 'was', would generate the output 'this ^ Awas^ Bis^ C a line'. The three control characters are used to delimit the new and old strings. Consumers of the output can use this information to determine actual source file column positions. The original characters reflect the state of the input line after trigraphs are substituted, continuation lines are catenated, and comments are removed. Use of these constructs preceding functional code on a line makes the column information inaccurate. Use of **-C** avoids this problem for comments.

-w

Prevents *cpp8086* from generating warnings.

Five special names are understood by *cpp8086*. They can be used anywhere (including in macros) just as any other defined name.

LINE is defined as the current line number (as a decimal integer) as known by *cpp8086*.

FILE is defined as the current file name (as a C string) as known by *cpp8086*.

DATE is defined as the current date (as a C string) of the form "Mmm dd yyyy".

TIME is defined as the current time (as a C string) of the form "hh:mm:ss".

STDC is defined as 1 indicating an ANSI standard C compiler.

All *cpp8086* directives start with lines begun by **#** . Any number of blanks and tabs are allowed before and after the **#** . The directives are:

 # define "name" " "token-string Replace subsequent instances of *name* with *token-string*. (*token-string* may be null).

 # define *name***(** *arg***, ...,** *arg* **)** *token-string* Notice that there can be no space between *name* and the **(**. Replace subsequent instances of *name(arg***, ...,** *arg)* by *token-string*, where each occurrence of an *arg* in the *token-string* is replaced by the corresponding set of tokens in the comma-separated list. When a macro with arguments is expanded, the arguments are placed into the expanded *token-string* after they have been recursively macro substituted. After the entire *token-string* has been expanded, *cpp8086* re-starts its scan for names to expand at the beginning of the newly created *token-string*. Any name which was expanded in a nested macro invocation is not available for expansion until the end of the parent macro.

The # operator in the replacement token-string is a "stringization" unary operator causing the parameter name following it to become a C string literal containing the substituted argument. For example:

```
# define stringize(a) # a
stringize(This will be a "string".\n)
```

becomes:

```
"This will be \"string\".\n"
```

The # # operator in the replacement token-string is a concatenation operator which allows the user to substitute for a portion of an identifier, operator, or other token by placing the # # between the parameter and the remainder of the token. First the parameter is substituted and then the # # and any white space surrounding it are removed. For example:

```
# define f(x) var # # x
f(3)
```

results in:

```
var3
```

 # undef "name"
Cause the definition of *name* (if any) to be forgotten from now on.

\# **include** *"filename"*
\# **include** < *filename*>
\# **include** *token-string*
Include at this point the contents of *filename* (which is then run through *cpp8086*). If the # **include** doesn’t match one of the first two forms then the *token-string* is macro substituted and retried to see if it matches one of the first two forms. See the **-I** option above for more detail.

\# **line** *integer-constant "filename"*
Causes *cpp8086* to generate line control information for the next pass of the C compiler. *Integer-constant* is the line number of the next line and *filename* is the file where it comes from. If *"filename"* is not given, the current file name is unchanged.

\# **endif**
Ends a section of lines begun by a test directive (# **if**, # **ifdef**, or # **ifndef**). Each test directive must have a matching # **endif**.

\# **ifdef "name"**
The lines following will appear in the output if and only if *name* has been the subject of a previous # **define** without being the subject of an intervening # **undef**.

\# **ifndef "name"**
The lines following will *not* appear in the output if and only if *name* has been the subject of a previous # **define** without being the subject of an intervening # **undef**.

\# **if "constant-expression"**
Lines following will appear in the output if and only if the *constant-expression* evaluates to non-zero. All binary non-assignment C operators, the **?:** operator, the unary **-**, **!**, and **~** operators are all legal in *constant-expression*. The precedence of the operators is the same as defined by the C language. There is also a unary operator **defined**, which can be used in *constant-expression* in these two forms: **defined (** *name* **)** or **defined** *name*. This allows the utility of # **ifdef** and # **ifndef** in a # **if** directive. Only these operators, integer constants, and names which are known by *cpp8086* should be used in *constant-expression*. In particular, the **sizeof** operator is not available.

\# **elif "constant-expression"**
Any number of # **elif** directives can occur between one of the test directives and the matching # **endif**. If none of the preceding test or # **elif** directives have been true and this *constant-expression* evaluates to true then the following

lines will appear in the output. The *constant-expression* is evaluated the same as in the **# if** directive.

 # else
Can occur after a test directive and any intervening **# elif** directives and before the matching **# endif** directive. If none of the preceding tests have been true then the following lines will appear in the output.

 # pragma
All lines with the **# pragma** directive are passed unchanged to the output except for removal of leading whitespace.

 # error "token-string"
Write a diagnostic message to stderr. The preprocessor will continue processing after this directive is encountered, but *cc8086* will not continue the compilation process. The **# error** directive is useful for debugging **# if** and **# ifdef** directives.

The test directives and the possible **# elif** and **# else** directives can be nested. The *cpp8086* command supports names up to 255 characters in length.

FILES

/usr/hp64000/include/8086 directory for standard **# include** files.

AUTHOR

The *cpp8086* command was developed by the Hewlett-Packard Company.

SEE ALSO

cc8086(1), m4(1)
B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, 1988
HP 8086/186 C Cross Compiler User's Guide, Hewlett-Packard, 1995

DIAGNOSTICS

The error messages produced by *cpp8086* are intended to be self-explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

NOTES

When new-line characters are found in argument lists for macros to be expanded, the current version of *cpp8086* replaces these new-lines with blanks.

clst8086 (1)

NAME

clst8086 - Listing generator for the 8086 C cross compiler

SYNOPSIS

/usr/hp64000/lib/clst8086 [options] outputfile

DESCRIPTION

The *clst8086* program is the listing generator of the C cross-compiler for the Intel 8086 microprocessor. It generates a listing in *outputfile* (*stdout*, if **-o** present) from the assembly source or listing file read from *stdin*. The lister's ability to include C source file lines is driven by **?file**, **?line**, and "*LINE*" directives in the assembly code. The preferred way to invoke *clst8086* is through the *cc8086*(1) command, since the functionality may someday be moved elsewhere. Therefore, the use of *clst8086* other than in this framework is not suggested.

The default listing produced contains C source lines with line numbers and interleaved assembly code (except special ?directives, see **-d** below). C include files are not expanded, and no cross reference is listed. The contents of listings generated vary with the following options:

The first two options support options of *cc8086*(1).

-i

Add C source included via **# include** C pre-processor directives.

-x

Add cross reference of symbols in C source and any expanded include files.

The following remaining options are not directly used by *cc8086*(1), but may be used via *cc8086*(1)'s **-W** option or by invoking the lister directly.

-a

Delete assembly source lines (possibly with associated program counter and object code values).

-c

Suppress the C source lines. This results in, essentially, an assembler listing or source file.

-d

Do not omit assembly source lines containing "?pseudo-operations".

-H *header*

Cause *header* to be used as the first line generated on each page of the listing.

-o

Override the output file specified and write the listing to *stdout*. This may be used, via **-W**, to cause *cc8086*(1) to produce listings to *stdout*.

-t

Surround C source statements with HP terminal "inverse video on" (i.e. esc & d B) and "inverse video off" (i.e. esc & d @) escape sequences for convenient terminal viewing. This is particularly handy when viewing lister output with the *more*(1) command; it is not too handy when viewing lister output with the *vi*(1) command.

Note that options other than the above are not recognized and cause a warning message to be written to *stderr*.

AUTHOR

Clst8086 was developed by the Hewlett-Packard Company.

SEE ALSO

as86(1), cc8086(1)
HP 8086/186 C Cross Compiler User's Guide, Hewlett-Packard, 1995

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.