DeSMET DC88
C COMPILER

MARK DeSMET

Published and Distributed by

C WARE CORPORATION
PASO ROBLES, CALIFORNIA

DeSmet C Development Package

Version 3.1 — May,1988

Version 3.03 — February, 1988 (DC88)
Version 3.0 — April, 1987
Version 2.5 — October, 1985
Version 2.4 — October, 1984
Version 2.3 — April, 1984

Published by: C Ware Corporation
P.O. Box 428
Paso Robles, CA 93447
USA

(805) 239-4620 (Tech Support/Sales)

(805) 239-4834 (Tech BBS)

Table of Contents

Introduction	
Overview	1.1
Large Case Option	1.3
Getting Started	
Backing Up	2.1
Installing The Software	2.1
Installing DC88	2.1
Installing DC88 on a Hard Disk	2.4
Installing DC88 on a Floppy Disk	2.7
Installing the Large Case Option	2.10
Installing Large Case on a Hard Disk	2.11
Installing Large Case on a Floppy Disk	2.12
A Short Example	2.13
Completion Codes	2.18
The SEE™ Text Editor	
Introduction	3.1
Getting Started	
Concepts	3.2
Starting the Editor	3.3
Inserting & Editing Text	3.4
Saving the File	3.9
Editing Existing Files	3.10
The Invocation Line	3.11
The Keyboard	
Cursor Movement Keys	3.12
Editing Keys	3.13
The DOS Key	3.14
Commands	3.15
Configuration	3.31

Table of Contents

The C88 C Compiler

Introduction	4.1
Invocation	4.1
Examples	4.3
The C Language	
Environment	
Character Set	4.4
Trigraph Sequences	4.4
Language	
Keywords	4.5
Identifiers	4.5
Floating constants	4.5
Integer constants	4.6
Character constants	4.6
String constants	4.6
Hardware data types	4.7
Enumerated type	4.8
Function prototyping	4.8
Preprocessor	
Conditional compilation	4.9
Source file inclusion	4.10
Macro replacement	4.11
Line control	4.11
Error	4.11
Pragma	4.11
Null	4.12
Predefined macros	4.12
Extensions	
Asm	4.12
Case range	4.13
Restrictions	
Forward references	4.14
Externs	4.14
Large Case Option	4.15

The ASM88 Assembler

Introduction	5.1
Invocation	5.1
Examples	5.2
Large Case ASM88	5.3

Table of Contents

The BIND Object File Linker

Introduction	6.1
Invocation	6.1
Examples	6.3
Small Case BIND	
Space Considerations	6.3
Overlays	6.4
Large Case BIND	6.6
Libraries	6.7

The LIB88 Object File Librarian

Introduction	7.1
Invocation	7.1
Examples	7.2
Libraries	7.2

The D88 C Language Debugger

Introduction	8.1
D88 Usage	8.1
Command Input	8.3
Expressions	8.3
Commands	8.5

Utility Programs

CLIST: a listing & xref utility	9.1
DUMP: a hex and ascii display utility	9.2
FASTSCR: a screen output enhancer	9.3
FREE: a free space display	9.3
GREP: a file search utility	9.3

Table of Contents

LS: a directory listing utility	9.4
MERGE: a C source and assembly language merge utility	9.5
MORE: a file screen listing utility	9.5
PCmake: a program maintainance utility	9.6
PROFILE: a performance monitor utility	9.9
RM: a file removal utility	9.11
SENSE87: an 8087/80287 sensing library	9.12
TOOLBOX.S: a library of useful tools	9.14
The CSTDIO Library	
Introduction	10.1
Names	10.1
Program Initialization	10.2
Calling Conventions	10.4
Memory Management	10.9
Input/Output Library	10.11
Directory Level Functions	10.11
File Level Functions	10.11
Stream Level Functions	10.12
Handle Level Functions	10.13
Screen Level Functions	10.13
Console Level Functions	10.14
Math Library	10.15
System Interface	10.16
Environment	10.18-1

Table of Contents

Library	10.18-2
Headers	
assert.h	10.18-2
ctype.h	10.18-2
math.h	10.18-3
setjmp.h	10.18-3
stdarg.h	10.18-3
stdio.h	10.18-4
stdlib.h	10.18-5
string.h	10.18-6
Functions & Macros	
Alphabetical by name	

Appendix A: Messages

ASM88 Messages	
Banner and Termination Messages	A.1
ASM88 Fatal Error Messages	A.2
ASM88 Error Messages	A.2
BIND Messages	
Banner and Termination Messages	A.7
BIND Fatal Error Messages	A.7
BIND Warning Messages	A.8
C88 Messages	
Banner and Termination Messages	A.9
C88 Fatal Error Messages	A.10
C88 Error Messages	A.12
C88 Warning Messages	A.16
C88 ASM88 Messages	A.17
CLIST Messages	
Banner and Termination Messages	A.18
CLIST Fatal Error Messages	A.18
D88 Messages	A.19
LIB88 Messages	
Banner and Termination Messages	A.21
LIB88 Fatal Error Messages	A.21
LIB88 Warning Messages	A.22
SEE Messages	
Banner and Termination Messages	A.23
SEE Error and Status Messages	A.23

Table of Contents

Appendix B: The ASM88 Assembly Language

Identifiers	B.1
Constants	B.1
Expressions	B.2
Registers	
General Registers	B.2
Byte Registers	B.3
Segment Registers	B.3
Addressing Modes	B.3
8086 Flags	B.4
Address Expressions	B.5
Address Typing	B.5
Comments	B.6
Assembler Direectives	B.6
Reserving Storage	B.7
Differences between ASM86 and ASM88	B.8
8086 Instructions	
Elements of Instructions	B.9
Instructions	B.9
8087	B.23
Control Word	B.24
Status Word	B.25
Tag Word	B.25
Condition Codes	B.26
8087 Instructions	B.27

Preface

This manual describes the DeSmet C Development Package for the IBM-PC personal computer and the other MS-DOS based personal computers. If you are unfamiliar with the C language or UNIX, the book *The C Programming Language* by Brian Kernighan and Dennis Ritchie is a good place to start. If you plan on coding in assembly language, it is advisable to get a manual on the Intel 8086 microprocessor. Books such as Intel's *ASM86 Language Reference Manual* or *The 8086 Family User's Guide* are good choices. These manuals fully describe the architecture and the instruction set of the 8086/8088 family of microprocessors.

We thank both the Pacific Data Works, and Scott Lewis for proofreading the many revisions of this manual.

Chapter 1

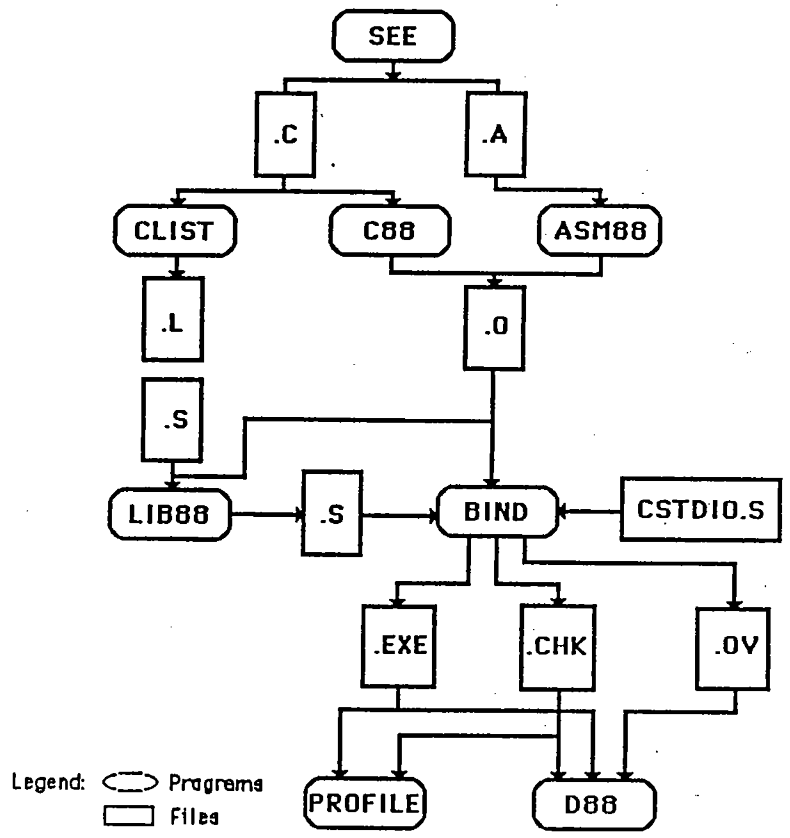
Introduction

Overview 1.1

Large Case Option 1.3

Overview

The DeSmet C Development Package is a set of programs and files for developing applications in the C programming language for the IBM-PC personal computer and its clones. The programs provided in this package require a minimum of 128K of Random Access Memory (RAM) and at least one disk drive. D88 requires 192K. Most programs will run under all versions of DOS, 1.xx, 2.xx, and 3.xx. The program execution profiler requires the use of DOS 2.x or later versions.



The diagram above outlines the interrelationships between some of the programs which are provided.

SEE is a full-screen, command oriented text editor designed for program editing rather than word processing. While SEE can edit any standand ASCII text file, its main purpose is to produce C [.C] and Assembler source files [.A]. The compiler C88 and the linker **BIND** can be invoked from SEE.

CLIST reads C source files [.C] and produces a listing file with a symbol cross-reference.

C88 is the C compiler. It reads C source files [.C] and produces either object files [.O] or assembler files [.A]. It supports the complete Kernighan and Ritchie C language plus the UNIX V7 extensions — structure assignment and parameter passing, and enumerated types. C88 supports both the Small and Large Case memory models.

ASM88 is the 8086/8088 assembler. It reads assembler source files [.A] and produces linkable object files [.O].

BIND is the object file linker. It reads object files [.O] and library files [.S] and produces an executable file [.EXE]. BIND optionally produces the debugger information file [.CHK] and overlay files [.OV]. The Large Case memory model linker is BBIND.

LIB88 is the object file librarian. It reads object files [.O] and other library files [.S] and produces library files [.S].

D88 is the C source-level symbolic debugger. It provides access to program variables by name, breakpoints by function name and line number, and special support for debugging interactive programs. Source code display and stepping by source lines are also supported.

PROFILE is the C program execution profiler. It monitors the execution of the application program and indicates where time is spent in the program.

CSTDIO.S is the Standard Library used by BIND to provide the Operating System and machine-level functions supported by the C language. Two libraries are provided in the development package, one that support the 8087 math coprocessor directly (CSTDIO7.S) and one that provides numeric support in software (CSTDIO.S). The Large Case memory model libraries are BCSTDIO.S and BCSTDIO7.S

Large Case Option

The Large Case Option addresses the needs of programs that fit neither the standard Small Case restrictions (64K of code, 64K of data *and* stack), the partitioning requirements of overlays, nor the communication limitations of the *exec* function. Its features include:

Full 1-megabyte addressability via 32-bit pointers.

Static variables combined within a single data-segment to speed access.

Large Case differs from Small Case in two aspects: pointers are four bytes long (segment:offset) rather than two bytes (offset), and function calls are inter-segment (segment:offset) instead of intra-segment (offset).

There are still some memory restrictions with Large Case. No derived data object — array or structure — may be larger than 64K. The total size of all `static` and global fundamental objects (`char, int, ...`) must be less than 64K. The restriction on `static` and global fundamental objects has to do with efficiency — they can be accessed with the same speed as Small Case.

Large Case programs are approximately 15 per-cent larger and slower than their Small Case equivalents.

WARNING: LOGIC ERRORS IN PROGRAMS USING 32-BIT POINTERS MAY BE HAZARDOUS TO YOUR COMPUTER!

Programs using 32-bit pointers can change any byte of memory via pointers. Thus, improperly initialized pointers can change critical portions of MSDOS, possibly causing corruption of, or damage to your DISKS.

In addition, corruption of the return address or function address can transfer control to an arbitrary location in memory, thereby activating code that may cause corruption of, or damage to your DISKS.

Chapter 2

Getting Started

Backing Up 2.1

Installing The Software

Installing DC88 2.1
Installing DC88 on a Hard Disk 2.4
Installing C88 on a Floppy Disk 2.7

Installing the Large Case Option 2.10
Installing Large Case on a Hard Disk 2.11
Installing Large Case on a Floppy Disk 2.12

A Short Example 2.13

Completion Codes 2.18

Backing Up

First things first. Copy all of the files from the distribution disks onto a set of working floppy diskettes or hard disk. The disks are not copy-protected so the DOS copy command can be used to copy the files. The package is distributed on three DOS 2 double-sided (360KB) or one DOS 3 quad (1.2MB) diskette.The distribution diskette(s) should never be used, they should be kept as the backup copy of the package.

Installing the Software

The following section assumes you have two drives: a floppy disk (drive A:) and either a hard disk (drive C:) or another floppy disk (drive B:). The system drive is the disk your machine "boots" from, either A: or C:. All of the relevant DeSmet C software is in the \DC88 sub-directory on the hard disk, and in the Root Directory on the floppy disk.

Installing DC88 — There is one information and six data files in the DC88 3.1 distribution. The files, and their contents are:

BIN.EXE An archive of executable files, containing

ASM88.EXE:	The 8088 assembler.
BIND.EXE:	The object file linker.
BUF128.EXE:	128 byte type-ahead buffer program.
BUGS!.EXE:	Arcade game (use 'BUGS! c' for color displays).
C88.EXE:	The first pass of the C compiler.
CLIST.EXE	The C listing and cross-reference utility.
COMPARE.EXE:	The source code comparison utility.
D88.EXE:	The C source-level symbolic debugger.
DUMP.EXE:	The hex file display utility.
FASTSCR.EXE	Screen output speed-up.
FREE.EXE	Disk free space display
GEN.EXE:	The second pass of the C compiler.
GREP.EXE	A file search utility
LIB88.EXE:	The object file librarian.
LIFE.EXE:	Full screen game of Life.
LS.EXE	A directory listing utility
MERGE.EXE	A C source and assembly language merge utility
MORE.EXE	A file listing utility
PCMAKE.EXE	A program maintenance utility
PROFEND.EXE:	Used by PROFILE.EXE.
PROFILE.EXE:	The program execution profiler.
PROFSTAR.EXE:	Used by PROFILE.EXE.
RAM.COM:	RAM Disk driver for DOS 2 and later systems.
RM.EXE	A file deletion utility
SEE.EXE:	The full-screen editor.
TOOBJ.EXE	.O to .OBJ converter.

GRAPHICS.EXE An archive of text and library files, containing

GRAPHICS.NEW	New release information
GRAPHICS.DOC	Graphics documentation
GRAPHICS.CGA	Small-case graphics for the CGA
GRAPHICS.HGA	Small-case graphics for the Hercules Adaptor

INCLUDE.EXE An archive of text files, containing

ASSERT.H	Diagnostic include file.
CTYPE.H	Character handling include file.
DOS.H	DOS function include file.
FLOAT.H	Floating-point constants include file.
LIMITS.H	Character and numerical limits include file.
MATH.H	Mathematics include file.
SETJMP.H	Non-local jump include file.
STDARG.H	Variable argument include file.
STDIO.H	Input/output include file.
STDLIB.H	General utility include file.
STRING.H	String handling include file.

LIB.EXE An archive of library files, containing

C88.LIB	Software F/P LINK library.
C887.LIB	8087 LINK library.
CSTDIO.S	Software F/P BIND library.
CSTDIO7.S	8087 BIND library.
LLINK.BAT	LINK typical batch file.
SENSE87.S	8087-sensing upgrades to CSTDIO.S
TOOLBOX.S	Utility function library.

OBJ.EXE An archive of object files, containing

C.OBJ	LINK start-off code.
COMPARE.O	Object Code form of comparison utility.
D88.O	Object version of D88 — part 1.
D88REST.O	Object version of D88 — part 2.
EXEC.O:	The Exec() and Chain() functions.
EXEC.OBJ	Object code for `exec()` and `chain()` functions.
MSVER1.O	Object code for DOS 1 I/O functions.
SEE.O	Object code of the SEE editor

SRC.EXE	An archive of source files, containing
BUF128.A:	Source code for BUF128.EXE.
C.ASM	Source code for runtime start-up function.
CB.C:	Source code for a brace matching program.
CLOCK.C	Source code to display clock face.
CONFIG.C	Source code for screen functions
DUMP.C:	Source code for DUMP.EXE.
FLIP.A	D88 screen Flip source code.
ISETUP.A	Source code for runtime start-up function.
LATER.C:	Source code for a file modification-date utility.
LIFE.C:	Source code for LIFE.EXE.
PCIO.A	INT 10H screen interface source code.
RUBRBAND.C	Line drawing source code.
STUB.ASM	LINK example source code.
TDRAW.C	Med-res drawing test.
TGETPUT.C	Screen area get/put test.
TXDRAW.C	High-res drawing test.
VERSION.DOC	Contains the latest information about the release and its contents.

If you have the 1.2MB disk format, all the files will be on the one disk. If you have the 360KB disk format, the files are on the following disks:

Disk #1	BIN.EXE, INCLUDE.EXE, and VERSION.DOC
Disk #2	GRAPHICS.EXE, LIB.EXE, and OBJ.EXE
Disk #3	SRC.EXE

Each of the archive files can extract some, or all, of its contents. For example, to extract all of the SRC.EXE archive file enter

```
src
```

To extract, say, just the PCIO.A file from the SRC.EXE archive, enter

```
src pcio.a
```

If the package is to be run on a system other than an IBM PC, XT, AT, PCjr or PC-clone, the screen interface for SEE must be configured before it can be used. See the notes in the file CONFIG.C in the SRC.EXE archive for details.

Installing DC88 on a Hard Disk.

1. For systems utilizing DOS 2 or later versions of the operating systems, make sure that the ASCII text file **CONFIG.SYS** exists in the Root Directory of your system disk (C:). If it doesn't exist, you can create it with SEE (If you don't know how to use SEE, look at the example in this chapter).

```
c:
cd \
see config.sys
```

The file must contain the line:

```
FILES=20
```

since DC88 supports 20 open files — stdin, stdout, stderr, stdaux, stdprt and 15 other files. The default number of eight is insufficient for the BIND program. If there is enough memory available, add the line:

```
BUFFERS=20
```

to improve file performance in the operating system. 512 bytes are allocated for each buffer specified.

If you have a system with more than 256KB of memory, then the Ram Disk driver RAM.COM can be used to create an extremely fast disk. To add the Ram Disk, extract RAM.COM from the BIN.EXE archive

```
a:bin ram.com
```

and add the line

```
DEVICE=RAM.COM n
```

to CONFIG.SYS. The parameter *n* is a decimal number between 32 and 650, indicating the size of Ram Disk in KB (1024 bytes) increments.

The Ram Disk installs as the next available drive — if the highest letter drive on your system was C:, then the Ram Disk will install as D:. Use the DOS chkdsk command to verify the drive assignment.

2. Create a sub-directory (i.e., \DC88) in the root directory of the hard disk (e.g., C:).

```
mkdir dc88
cd dc88
```

3. Unpack the BIN, INCLUDE, LIB and OBJ archives to DC88.

   - Disk #1 — 1.2MB & 360KB format.

     ```
     a:bin c88.* gen.* asm88.* bind.* d88.* see.*
     a:include
     ```

   - If you wish to use LINK

     ```
     a:bin toobj.exe
     ```

   - If you have the 360KB format, insert Disk #2 in drive A:

   - If you wish to create programs that use <u>only</u> hardware F/P

     ```
     a:lib cstdio7.s
     ren cstdio7.s cstdio.s
     ```

     else, if you wish to create programs that use <u>only</u> software F/P

     ```
     a:lib cstdio.s
     ```

     else, if you wish to create programs that use either F/P

     ```
     a:lib cstdio.s sense87.s
     ren *.s *.o
     lib88 sense87 cstdio -ocstdio
     del cstdio.o
     del sense87.o
     ```

   - If you wish to use LINK

     ```
     a:obj c.obj exec.obj
     ```

     If you wish to create programs that use <u>only</u> hardware F/P

     ```
     a:lib c887.lib
     ```

     else, if you wish to create programs that use <u>only</u> software F/P

     ```
     a:lib c88.lib
     ```

Be sure to change the Bind Flags in SEE (using the SET command) to invoke LINK instead of BIND, or use the LLINK.BAT file as model for linking.

- If you want your library to use only DOS 1 functions

```
a:obj msver1.o
ren cstdio.s cstdio.o
lib88 msver1 cstdio -ocstdio
del cstdio.o
del msver1.o
```

3. If you wish to use the Graphics Package, print the manual and text

```
a:graphics graphics.doc graphics.new
copy graphics.* prn
del graphics.*
```

If you have a Color Graphics Adaptor (CGA), extract its library

```
a:graphics graphics.cga
ren graphics.cga libg.s
```

If you have a Hercules Adaptor (HGA), extract its library

```
a:graphics graphics.hga
ren graphics.hga libg.s
```

4. If you have a machine other than an IBM or close clone copy.

```
a:obj see.o d88.o d88rest.o compare.o
```

If you have the 360KB format, insert Disk #3.

If your machine emmulates the IBM ROM BIOS interrupt 10H, then recreate SEE, D88, & COMPARE

```
a:src pcio.a
asm88 pcio
bind see pcio -osee
bind d88 d88rest pcio -od88
bind compare pcio -ocompare
```

otherwise modify CONFIG.C for your particular display, then recreate SEE, D88, & COMPARE

```
a:src config.c
edit config.c
c88 config
bind see config -osee
bind d88 d88rest config -od88
bind compare config -ocompare
```

Delete `see.o`, `d88.o`, `d88rest.o`, and `compare.o`.

5. Modify AUTOEXEC.BAT to specify the location of DC88 components and include files.

   ```
   see \autoexec.bat
   ```

   The DC88 components are specified in the PATH environment variable. Add the **c:\dc88** sub-directory to the existing PATH specification, or create a PATH specification. See your DOS manual for information on specifying the PATH variable.

   The DC88 include files are specified in either the DSINC or the INCLUDE environment variable. Add either the **set DSINC=c:\dc88\** or the **set INCLUDE=c:\dc88\** line to the AUTOEXEC.BAT file. See Chapter 4 — The C88 C Compiler — for more information on the specifying the search path for DC88 include files.

6. Re-boot the system.

Installing DC88 on a Floppy Disk

1. Create a System Disk on drive B:

   ```
   format b:/s
   copy format.com b:
   ```

2. Put the System Disk in drive A: and DC88 Disk #1 in drive B: For systems utilizing DOS 2 or later versions of the operating systems, create the ASCII text file **CONFIG.SYS** in the Root Directory of your system disk (A:). You can create it with SEE (If you don't know how to use SEE, look at the example in this chapter).

   ```
   b:see config.sys
   ```

The file must contain the line:

```
FILES=20
```

since DC88 supports 20 open files — stdin, stdout, stderr, stdaux, stdprt and 15 other files. The default number of eight is insufficient for the BIND program. If there is enough memory available, add the line:

```
BUFFERS=20
```

to improve file performance in the operating system. 512 bytes are allocated for each buffer specified.

If you have a system with more than 256KB of memory, then the Ram Disk driver RAM.COM can be used to create an extremely fast disk. To add the Ram Disk, extract RAM.COM from the BIN.EXE archive

```
b:bin ram.com
```

and add the line

```
DEVICE=RAM.COM n
```

to CONFIG.SYS. The parameter *n* is a decimal number between 32 and 650, indicating the size of Ram Disk in KB (1024 bytes) increments.

The Ram Disk installs as the next available drive — if the highest letter drive on your system was B:, then the Ram Disk will install as C:. Use the DOS chkdsk command to verify the drive assignment.

2. Unpack the BIN, INCLUDE, LIB and OBJ archives to the system disk.

   - Disk #1 — 1.2MB & 360KB format.

     ```
     b:bin c88.* gen.* asm88.* bind.* d88.* see.*
     b:include
     ```

   - If you wish to use LINK

     ```
     b:bin toobj.exe
     ```

   - If you have the 360KB format, insert Disk #2 in drive B:

- If you wish to create programs that use <u>only</u> hardware F/P

  ```
  b:lib cstdio7.s
  ren cstdio7.s cstdio.s
  ```

  else, if you wish to create programs that use <u>only</u> software F/P

  ```
  b:lib cstdio.s
  ```

  else, if you wish to create programs that use either F/P

  ```
  b:lib cstdio.s sense87.s
  ren *.s *.o
  lib88 sense87 cstdio -ocstdio
  del cstdio.o
  del sense87.o
  ```

- If you wish to use LINK

  ```
  b:obj c.obj exec.obj
  ```

  If you wish to create programs that use <u>only</u> hardware F/P

  ```
  b:lib c887.lib
  ```

  else, if you wish to create programs that use <u>only</u> software F/P

  ```
  b:lib c88.lib
  ```

  Be sure to change the Bind Flags in SEE (using the SET command) to invoke LINK instead of BIND, or use the LLINK.BAT file as model for linking.

- If you want your library to use only DOS 1 functions

  ```
  b:obj msver1.o
  ren cstdio.s cstdio.o
  lib88 msver1 cstdio -ocstdio
  del cstdio.o
  del msver1.o
  ```

3. If you wish to use the Graphics Package, print the manual and text

   ```
   b:graphics graphics.doc graphics.new
   copy graphics.* prn
   del graphics.*
   ```

If you have a Color Graphics Adaptor (CGA), extract its library

```
b:graphics graphics.cga
ren graphics.cga libg.s
```

If you have a Hercules Adaptor (HGA), extract its library

```
b:graphics graphics.hga
ren graphics.hga libg.s
```

4. If you have a machine other than an IBM or close clone copy.

```
b:obj see.o d88.o d88rest.o compare.o
```

If you have the 360KB format, insert Disk #3.

If your machine emmulates the IBM ROM BIOS interrupt 10H, then recreate SEE, D88, & COMPARE

```
b:src pcio.a
asm88 pcio
bind see pcio -osee
bind d88 d88rest pcio -od88
bind compare pcio -ocompare
```

otherwise modify CONFIG.C for your particular display, then recreate SEE, D88, & COMPARE

```
b:src config.c
edit config.c
c88 config
bind see config -osee
bind d88 d88rest config -od88
bind compare config -ocompare
```

Delete `see.o`, `d88.o`, `d88rest.o`, and `compare.o`.

5. Re-boot the system.

Installing the Large Case Option

The Large Case Option is distributed on a single 5 1/4 inch floppy diskettes, containing:

B88.LIB:	Large Case DOS LINK C Library (non-8087)
B887.LIB:	Large Case DOS LINK C Library (8087)
BBIND.EXE:	Large Case Binder.

BC.ASM	Large Case DOS LINK start-up source code
BC.OBJ	Large Case DOS LINK start-up object code
BCSTDIO.S	Large Case C Library (non-8087)
BCSTDIO7.S	Large Case C Library (8087)
BEXEC.O	Large Case `exec()` and `chain()` functions.
BEXEC.OBJ	Large Case DOS LINK `exec()` and `chain()` functions.
BGRAPHIC.CGA	Large-case graphics for the CGA
BGRAPHIC.HGA	Large-case graphics for the Hercules Adaptor
BLLINK.BAT	Large Case DOS LINK
BSTUB.ASM	Large Case DOS LINK MASM example

Installing Large Case on a Hard Disk

Place the Large Case Option disk in drive A:

```
    copy a:*.exe c:\dc88
```

If you have a 8087 coprocessor

```
    copy a:bcstdio7.s c:\dc88\bcstdio.s
```

otherwise

```
    copy a:bcstdio.s c:\dc88
```

If you are using LINK

```
    copy a:*.obj c:\dc88
```

If you have a 8087 coprocessor

```
        copy a:bc887.lib c:\dc88\bc88.lib
```

otherwise

```
        copy a:bc88.lib c:\dc88
```

Be sure to change the Bind Flags in SEE (using the SET command) to invoke BBIND or LINK instead of BIND, or use the BLLINK.BAT file as model for linking.

If you are using the Graphics Package with a Color Graphics Adaptor

```
    copy a:bgraphic.cga c:\dc88\blibg.s
```

otherwise

```
    copy a:bgraphic.hga c:\dc88\blibg.s
```

Installing Large Case on a Floppy Disk

Place the Large Case Option disk in drive B: and the DC88 System Disk in drive A: and copy the following files:

```
    copy b:*.exe
```

If you have a 8087 coprocessor

```
    copy b:bcstdio7.s bcstdio.s
```

otherwise

```
    copy b:bcstdio.s
```

If you are using LINK

```
    copy b:*.obj a:
```

If you have a 8087 coprocessor

```
        copy b:bc887.lib bc88.lib
```

otherwise

```
        copy b:bc88.lib
```

Be sure to change the Bind Flags in SEE (using the SET command) to invoke BBIND or LINK instead of BIND, or use the BLLINK.BAT file as model for linking.

If you are using the Graphics Package with a Color Graphics Adaptor

```
    copy b:bgraphic.cga blibg.s
```

otherwise

```
    copy b:bgraphic.hga blibg.s
```

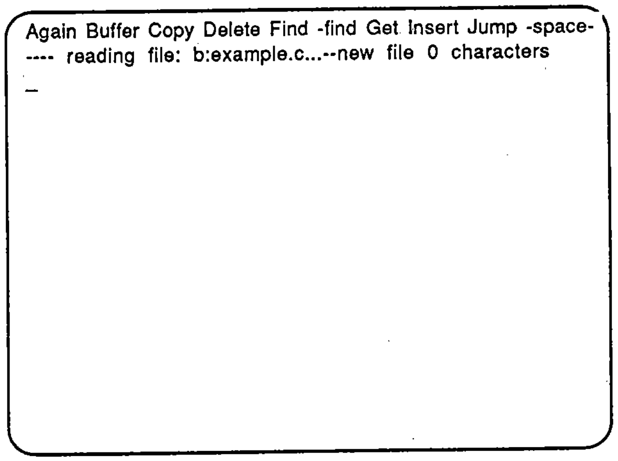
A Short Example

This example shows the general method for creating executable programs with this package. It assumes that the disk in the default drive, in this case drive A:, contains the compiler (C88.EXE and GEN.EXE), the assembler (ASM88.EXE), the binder (BIND.EXE), the standard library (CSTDIO.S) and the text editor (SEE.EXE). The source code will reside on drive B:.

Enter the example program with the SEE text editor. To start the SEE text editor, type:

```
see  b:example.c
```

The screen will look as follows:



```
Again Buffer Copy Delete Find -find Get Insert Jump -space-
---- reading file: b:example.c...--new file 0 characters
_
```

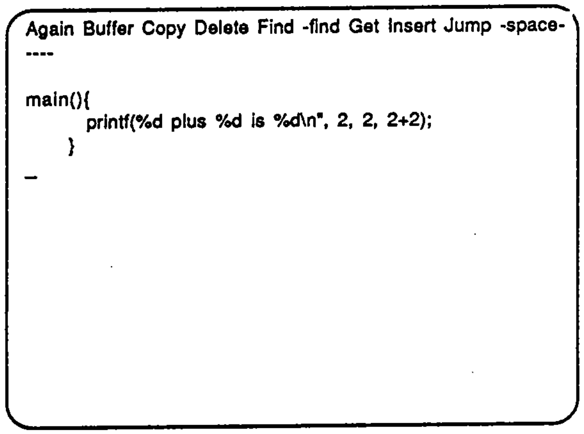
Figure 2-1
See™ Initial Screen

Other than the header, footer, and this sentence, this page is intentionally blank.

Type the letter 'I', or press the 'Ins' key, to put the editor into Insert mode. Now type in the following program:

```
main() {<Ret>
<Tab>printf("%d plus %d is %d\n", 2, 2, 2+2);<Ret>
<Tab>}<Ret>
<Esc>
```

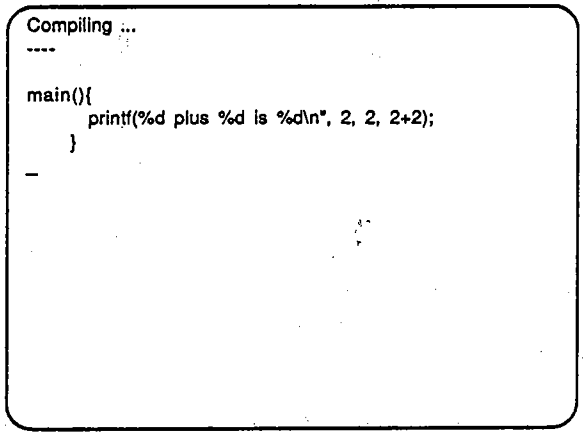
Note that the items <Tab>, <Ret>, and <Esc> indicate the Tab, Return, and Esc keys, respectively. The <Esc> will terminate insert mode and return the editor to command mode. The screen should now appear as follows:



```
Again Buffer Copy Delete Find -find Get Insert Jump -space-
----

main(){
        printf(%d plus %d is %d\n", 2, 2, 2+2);
    }
_
```

Figure 2-2
Program Display

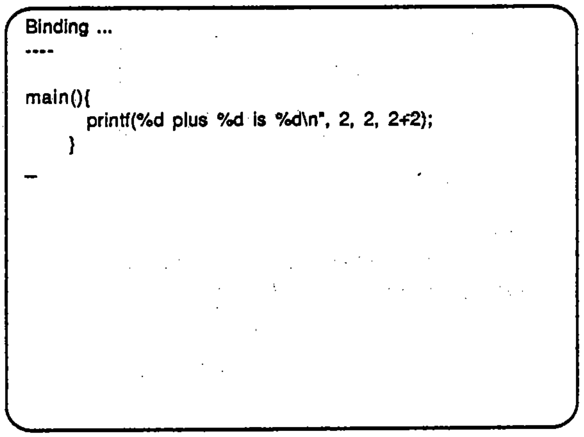
To compile the program just entered, type the sequence of characters, 'Q' for Quit and 'C' for Compile. This will start C88 using the file in memory. The computer will display:



```
Compiling ...
----

main(){
        printf(%d plus %d is %d\n", 2, 2, 2+2);
      }
_
```

Figure 2-3
Compiling from SEE

The message "Compiling ..." replaces the first line of the display. If there are errors during the compilation, the error message will appear on the second line of the display, and the cursor will be on the error line. You can correct the error and recompile. If there are no errors BIND will be invoked. The screen appears as follows:



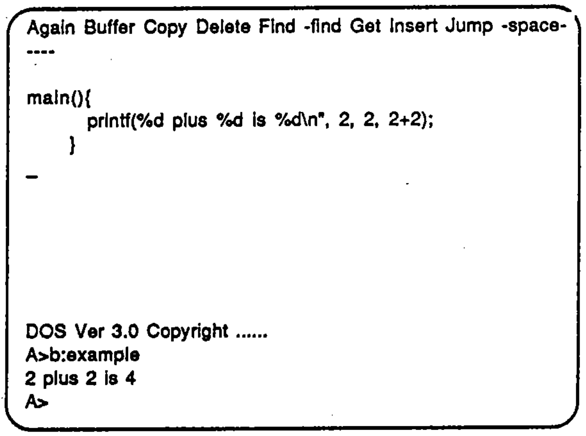
```
Binding ...
----

main(){
        printf(%d plus %d is %d\n", 2, 2, 2+2);
      }
_
```

Figure 2-4
Binding from SEE

If there are any errors, they will be displayed on the message line. To run the program, press esc to escape from the Quit menu and press the F9 key to invoke a new DOS shell. At the prompt, enter

```
b:example
```

to invoke the program. The screen will look something like:



```
Again Buffer Copy Delete Find -find Get Insert Jump -space-
----

main(){
        printf(%d plus %d is %d\n", 2, 2, 2+2);
      }
-






DOS Ver 3.0 Copyright ......
A>b:example
2 plus 2 is 4
A>
```

Figure 2-5
Executing example Program

To return to SEE type

```
exit
```

at the DOS propmpt. You will be returned to the SEE display.

If you wish to save the file to disk, type 'Q' (Quit) followed by an 'S' (Save-exit). The file will be saved, and control will be returned to DOS.

Completion Codes

The C88, ASM88, BIND and LIB88 programs set the completion code to:

0	if no warnings or errors occurred,
1	if warnings were issued, and
2	if errors occurred.

Batch files can take advantage of these values to stop execution or otherwise handle these exceptional cases.

The batch file CC.BAT listed below will stop if C88 or BIND reports an error:

```
c88 %1
if errorlevel 1 goto stop
bind %1
if errorlevel 1 goto stop
%1
:stop
```

More complicated development situations can be handled with the program LATEP which is supplied in source form in the file LATER.C. LATER takes a list of filenames as arguments. It sets the errorlevel to one if the last file does not exist or if the last file has an earlier modification date than any other file in the list. It can only be used on systems with a battery backup clock or where users are careful about setting the date and time when the system is brought up. Assume a program is composed of the files moda.c, modb.c, modc.c and the include file mod.h. The following .BAT file can be used to regenerate the program whenever a module changes:

```
later moda.c mod.h moda.o
if errorlevel 1 c88 moda
if errorlevel 1 goto stop
later modb.c mod.h modb.o
if errorlevel 1 c88 modb
if errorlevel 1 goto stop
later modc.c mod.h modc.o
if errorlevel 1 c88 modc
if errorlevel 1 goto stop
later moda.o modb.o modc.o mod.exe
if errorlevel 1 bind moda modb modc -omod
:stop
```

This provides a service similar to the UNIX MAKE program. Only those files that need to be compiled will be compiled.

Chapter 3

The SEE™ Text Editor

Introduction	3.1
Getting Started	
Concepts	3.2
Starting the Editor	3.3
Inserting and Editing Text	3.4
Saving the File	3.9
Editing Existing Files	3.10
The Invocation Line	3.11
The Keyboard	
Cursor Movement Keys	3.12
Editing Keys	3.13
The DOS Key	3.14
Commands	3.15
Again	3.16
Buffer	3.16
Copy	3.16
Delete	3.16
Find	3.17
-Find	3.18
Get	3.18
Insert	3.18
Jump	3.18
List	3.19
Macro	3.19
Other	3.20
Put	3.20
Quit	3.20
Replace	3.23
Set	3.23
Tag	3.30
Version/View	3.30
Wrap	3.30
Xchange	3.30

Commands (cont.)	
#	3.31
{}()[]	3.31
\	3.31
Configuration	3.31

Introduction

SEE is a general purpose full-screen text editor designed for program entry rather than word processing. It features:

- invoking the compiler (C88) and the binder (BIND) from the editor — errors return control to the editor at the error line,
- invoking a copy of the shell (COMMAND.COM) to provide access to DOS functions,
- handling files larger than available memory,
- editing two files simultaneously,
- viewing the two files either on separate screens, or in two windows on the same screen,
- a macro facility which allows you to capture a series of keystrokes and replay them to ease repetitive tasks,
- automatic indentation,
- brace/bracket/parenthesis matching to ease program entry,

SEE is shipped configured for the IBM-PC and its clones. SEE may be reconfigured to run on other machines which support DOS but have different keyboard and/or screen interfaces than the IBM-PC (see Section 3.6).

Getting Started

Concepts

SEE does not directly manipulate a file on the disk. It brings a copy of the file into memory and performs all work on this internal copy. The file on the disk is not modified until the copy in memory is stored on the disk. If the file is larger than the internal buffer area, SEE will open "spill" files to swap the edited text in and out of memory. For this reason, you should not have any files named SEETMP.###, where ### is a series of three digits (currently restricted to 000, 001, 002, 003, and 004).

Commands are executed by typing the first letter of the command displayed on the menu line (the first line on the screen). For example, to execute the Delete command, simply type the letter 'D'; the case of the letter does not matter.

Whenever a <u>block</u> of text is deleted with the Delete command, the text is placed in a special area known as the <u>copy buffer</u>. Blocks selected with the Buffer command are also placed in this buffer. When the Copy command is used, the contents of this buffer is inserted into the text at the cursor location. The copy buffer is maintained as long as the editor is running and is shared by both files (if two files are being edited). This is the mechanism used to move text from one location to another or from one file to another.

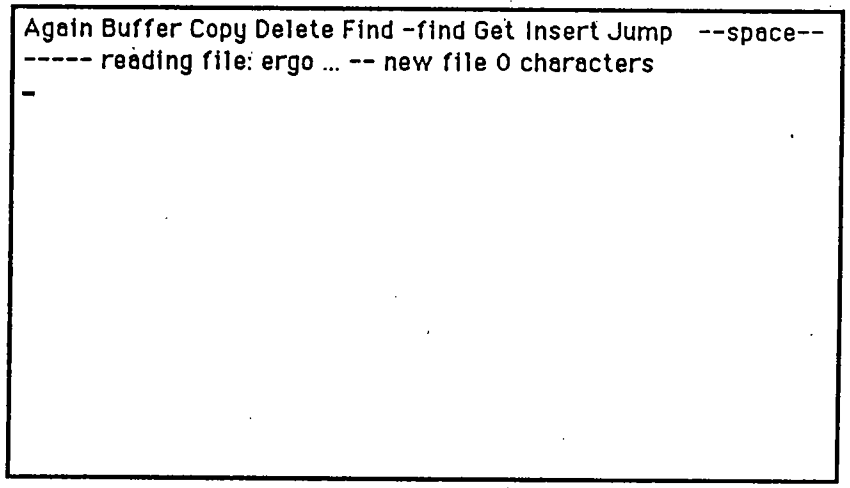
The cursor indicates the location where all action will occur. It will be in one of three states: a double-bar cursor indicating command mode, a single-bar cursor indicating Insert mode or a block cursor indicating Exchange mode. The cursor is always visible on the screen. As the cursor is moved to an edge of the screen, the screen will scroll the text to keep the cursor in view, both vertically and horizontally. For example, if the cursor is moved down when it is on the last line of the screen, the screen will be scrolled up one line to show the line the cursor is on. Similarly, when the cursor is in the rightmost column of the screen and the cursor is moved to the right (assuming the line has more characters not currently displayed on the screen), the screen will be scrolled to the left by 15 columns to show the new location.

Starting the Editor

To start the editor to edit a new file named 'ergo', simply type:

see ergo

and the computer should respond with the screen:



```
Again Buffer Copy Delete Find -find Get Insert Jump   --space--
----- reading file: ergo ... -- new file 0 characters
-
```

The top line on the display is the menu line. This line displays the current mode of the editor and the commands available at any given time. In this first screen, the menu line contains the first set of commands available at the command level:

```
Again Buffer Copy Delete Find -find Get Insert Jump   --space--
```

Hitting the space bar displays the second set of commands:

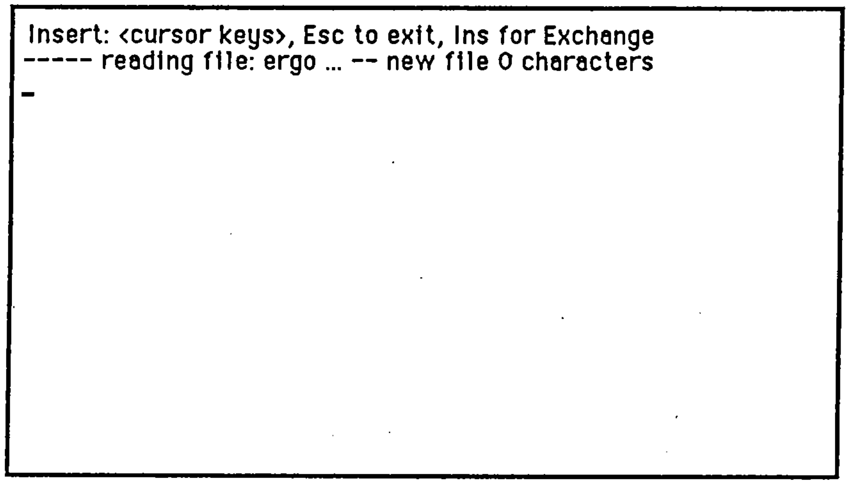
```
List Macro Other Put Quit Replace Set Tag Wrap Xchange --space--
```

Hitting the space bar again will redisplay the first set of commands. The commands are fully described in Section 3.5 of this manual. Each command may be executed by typing the first letter of its menu item; for example, A for Again, B for Buffer, etc. The case of the command letter is not important.

The second line of the screen is used to display messages and status from the various commands and is naturally called the message line. The message "ergo ... -- new file 0 characters" indicates that the file ergo has not been found and that the internal file buffer is empty.

Inserting and Editing Text

To insert text into the file, we must enter Insert mode. Do this by either typing the letter 'I' to execute the insert command, or by pressing the Ins key. The screen should now look as follows:



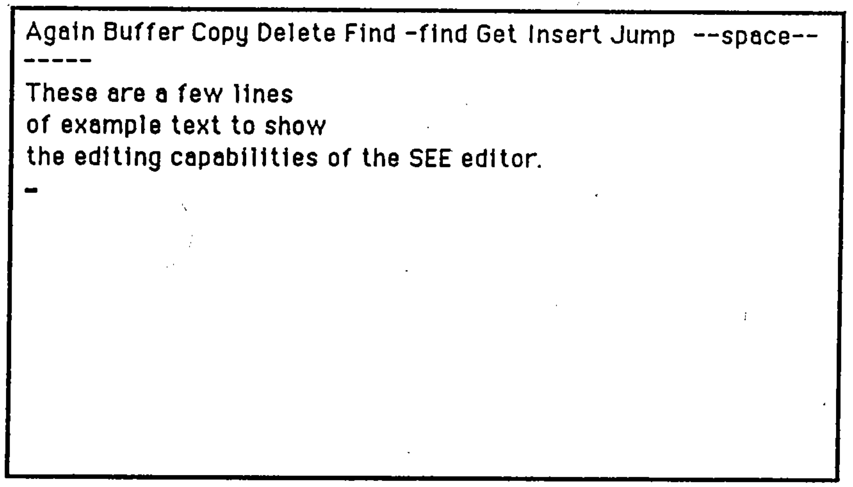
```
Insert: <cursor keys>, Esc to exit, Ins for Exchange
----- reading file: ergo ... -- new file 0 characters
-
```

Note that the menu line has changed to indicate the types of actions, other than inserting text, that may be performed. Any character now typed, except for one of the special keys described in Section 4, will now be inserted into the text at the cursor location, just prior to the character that the cursor is on.

Now type in the lines:

These are a few lines <Return>
of example text to shoe<Backspace>w<Return>
the editing capabilities of the SEE editor. <Return>
<Esc>

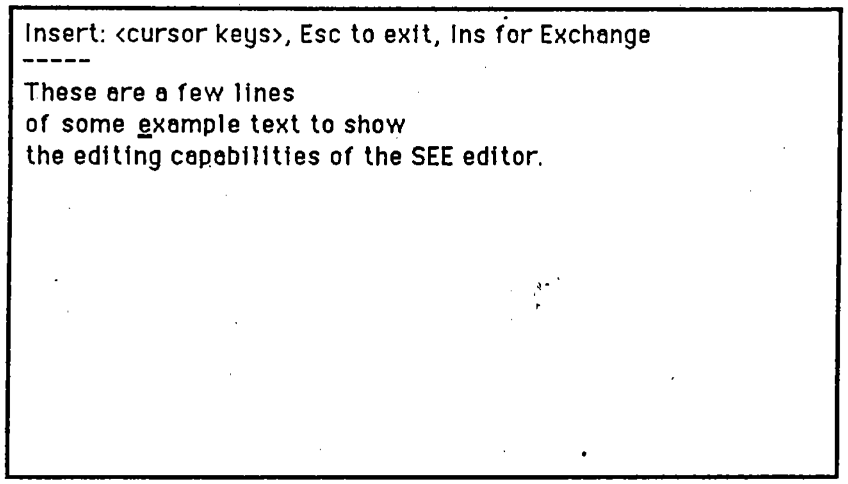
The screen should now look as follows:



```
Again Buffer Copy Delete Find -find Get Insert Jump  --space--
-----
These are a few lines
of example text to show
the editing capabilities of the SEE editor.
-
```

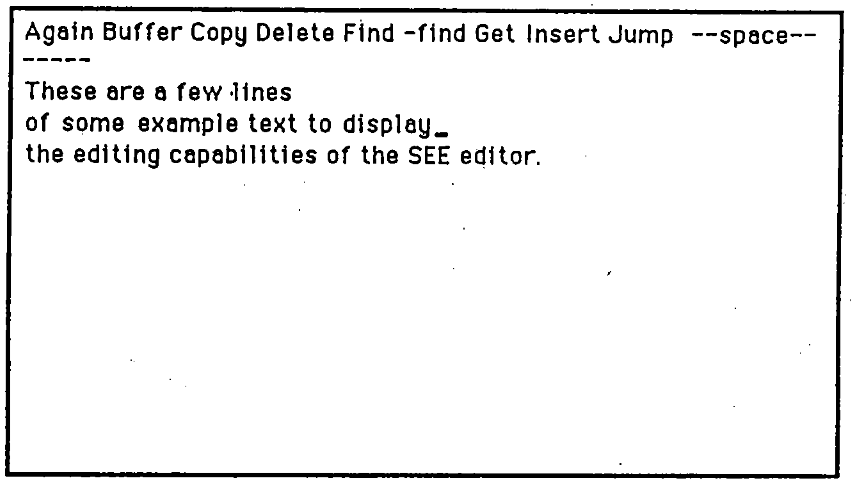
Note that the symbols <Return>, <Backspace>, and <Esc> represent the use of the return, backspace, tab, and Esc keys, respectively. The <Return> inserts a carriage-return, line-feed (CRLF) sequence into the file to begin a new line and the cursor moves down one line and to the left side of the screen. The <Backspace> key deletes the character preceding the cursor. The <Tab> key inserts a tab character into the file which is expanded to the next tab stop. Tab stops, by default, are located every four characters, however this value may be changed in the Set command. The <Esc> key breaks the editor out of Insert mode and places it back in command mode.

The cursor keys are used to move the cursor around the screen in small increments. Press the up-arrow key twice to move the cursor up to the beginning of the second line. Press the right-arrow key three times to move the cursor to the beginning of the word 'example'. Type the letter 'I' to put the editor into Insert mode and type the word 'some' without the quotes and add a blank. Note that as each character is typed, the rest of the line is "pushed" to the right. The screen should now look as follows:



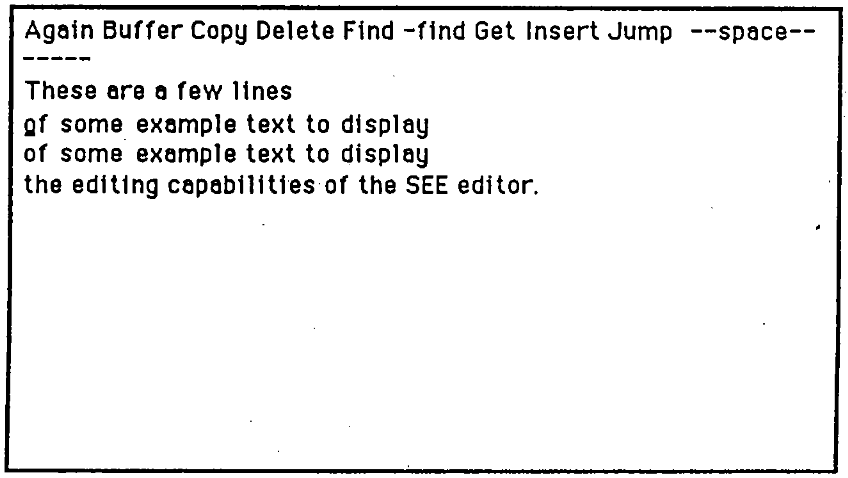
```
Insert: <cursor keys>, Esc to exit, Ins for Exchange
-----
These are a few lines
of some example text to show
the editing capabilities of the SEE editor.
```

Now hold down the control key (Ctrl) and press the right-arrow key three times. Note that the cursor jumps from one word to the next when using this combination of keys. See Section 4 for full details on all of the special keys. Also note that the editor does not have to be in command mode to use the cursor movement keys. Now hit the Ins key to change from Insert mode to Exchange mode; the menu line will display Exchange instead of Insert. In Exchange mode, the character at the cursor is overwritten by the new character rather than having the character inserted into the file. The only exception to this rule is when the cursor is positioned at the end of a line, characters are inserted rather than overwriting the CRLF end-of-line sequence. Exchange mode can also be entered from command mode by typing the letter 'X' for Xchange. Type the word 'display' and notice how the word 'show' is overwritten with the new word 'display'. Press the Esc key to go back to command mode. The screen should now look as follows:



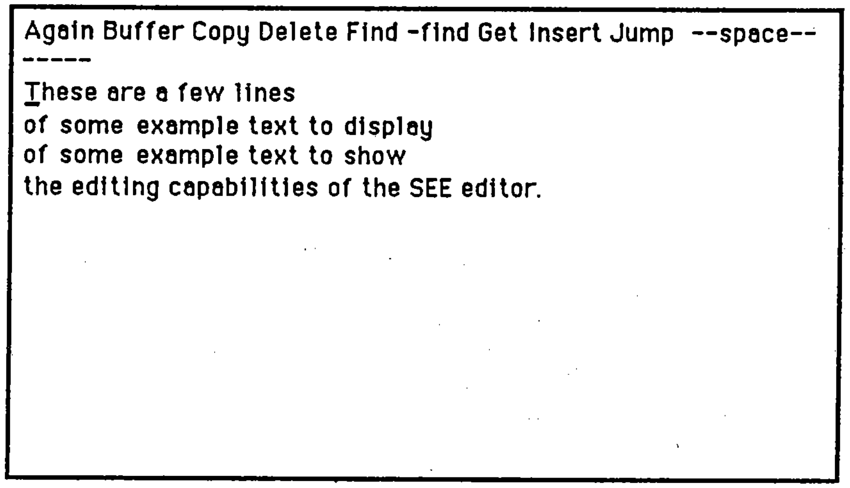
```
Again Buffer Copy Delete Find -find Get Insert Jump --space--
-----
These are a few lines
of some example text to display_
the editing capabilities of the SEE editor.
```

Press the Home key and note the location of the cursor. To delete this line, invoke the Delete command by typing the letter 'D', move the cursor down one line with the down-arrow key, and type the letter 'D' again to complete the deletion (the Esc key will also work). The second line has been deleted and placed in the copy buffer. Now type the letter 'C' to invoke the Copy command to retrieve the text that was deleted. Type the letter 'C' again and a second copy of the line is inserted. The copy buffer always contains the last Deleted or Buffered block of text. The screen should now look as follows:



```
Again Buffer Copy Delete Find -find Get Insert Jump --space--
-----
These are a few lines
of some example text to display
of some example text to display
the editing capabilities of the SEE editor.
```

To find the first occurrence of the word 'display', press the letter 'F' to invoke the Find command. Type in the word 'display' (without the quotes) and press either Esc or Return to begin the search. The cursor should now be positioned after the word 'display' on the second line. To replace the next occurrence of the word 'display' with the word 'show', press the letter 'R' to invoke the Replace command. Notice that the previous search string 'display' now appears on the message line. Since this is the string to be replaced, simply press the Esc or Return key to select the string (rather than retyping the string). Type in the string 'show' and hit the Esc or Return key to execute the command. Press the Home key twice to move the cursor to the top of the screen. The screen should now appear as follows:



```
Again Buffer Copy Delete Find -find Get Insert Jump --space--
-----
These are a few lines
of some example text to display
of some example text to show
the editing capabilities of the SEE editor.
```

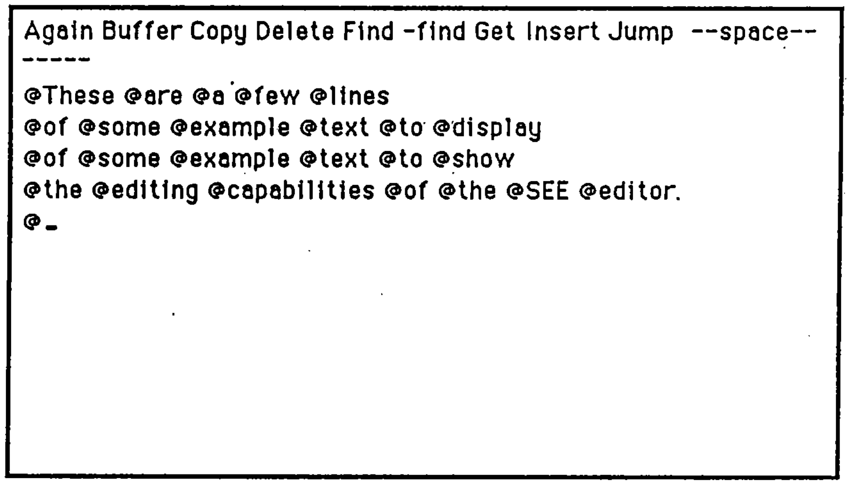
Another useful feature in SEE is its ability to record a series of keystrokes, command, cursor keys, etc., and replay them on command. These recordings are called **macros**. To create a macro, type 'M' to invoke the Macro command, type 'R' to indicate that a recording is to be made, and select the function key (F1 through F8) that is to be used to invoke the macro. In this example, press the F1 key. The message line now displays the line:

```
recording Macro F1, use Macro key to complete recording
```

This message will be displayed after every command to indicate that a macro recording is in progress. Now, any commands or special keys typed will be recorded into the macro until the Macro command is executed once again: For this example, execute the following commands:

```
I@<Esc><control right-arrow>M
```

Macro F1 is now defined to insert the '@' character in front of each word. To execute the macro, press the F1 key. To execute the macro a fixed number of times, say five times, type the number 5 and then the function key F1. The macro is executed five times. To execute the macro for the rest of the words in the file, type in a large number or use the more convenient '/' character to indicate the number 32767, the largest number. Type '/' and press the F1 key. The screen should now appear as follows:



```
Again Buffer Copy Delete Find -find Get Insert Jump --space--
-----
@These @are @a @few @lines
@of @some @example @text @to @display
@of @some @example @text @to @show
@the @editing @capabilities @of @the @SEE @editor.
@_
```

Saving the File

Recall that all of the editing was performed on the file in memory. This copy of the file must be written out to the disk. Type the letter 'Q' to enter the Quit menu. The choices under the Quit menu are:

```
BAKup  Compile  Exit  Initialize  Save-exit  Update  Write
```

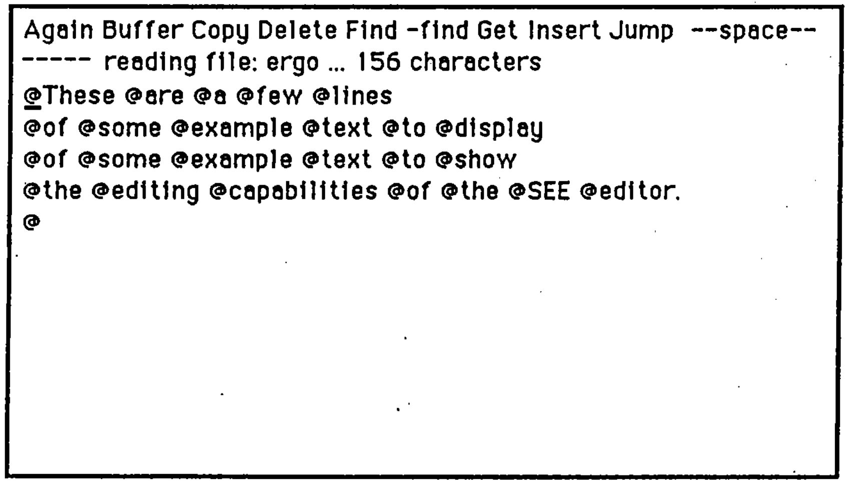
Each menu item is explained in detail in Section 5 under the Quit command. Press the letter 'S' to save the memory copy of the file to the disk file named 'ergo' which was entered at the beginning of this example. This selection will also terminate the editor.

Editing Existing Files

Now to edit the file ergo again, simply type the line:

see ergo

The editor will be loaded and will attempt to load the file ergo. If the file was loaded correctly, the screen should appear as follows:



```
Again Buffer Copy Delete Find -find Get Insert Jump  --space--
----- reading file: ergo ... 156 characters
@These @are @a @few @lines
@of @some @example @text @to @display
@of @some @example @text @to @show
@the @editing @capabilities @of @the @SEE @editor.
@
```

Type 'Q' to select the Quit command and then type 'E' to exit from the editor without writing the file out, since nothing has changed.

You now have a basic understanding of how to edit files with the SEE editor. Practice editing other files using the skills developed in this example. Don't be afraid to experiment. Remember that as long as you don't write the file back out to the disk, the old copy is safe. When you are comfortable with these editing features, look through the rest of the manual to see what else can be done and experiment with some new features.

The Invocation Line

There are a few different options available when starting the SEE editor. Invoking SEE with the command line:

```
see
```

will bring up the editor with an empty buffer and no filename specified. To save the file to disk, use the Write option under the Quit command described in Section 5.

Invoking SEE with the command line:

```
see <filename>
```

will have the editor load the file <filename> if it exists. <filename> will be used by the Update and BAKup options in the Quit command. If the file doesn't exist, SEE will act as if it existed but was a zero length file. Note that the file is not created until it is written out to disk.

Invoking the editor with the command line:

```
see <filename1> <filename2>
```

will have the editor load the text from <filename1> but will write out the text to <filename2>. <filename1> will not be altered by the edit session.

Adding the `-l` option to the command line:

```
see <filename> -lnnnn
```

will have the editor load the text from <filename1> and start editing at line *nnnn* . The `-l` option works with either filename configuration.

The Keyboard

This section describes the special keys used by the SEE editor as defined for the IBM-PC keyboard. If the editor has been reconfigured for a different keyboard, you will have to map the reconfigured keys to the IBM-PC keys to understand the following documentation.

Cursor Movement Keys

In the following descriptions, the caret (^) preceding the name of the key indicates that the control (Ctrl) key must be held down while the key is pressed.

Home:	When the Home key is pressed once, the cursor will move to the beginning of the current line (the line that the cursor is currently on). If the Home key is pressed twice in succession, the cursor will move to the beginning of the first line on the screen.
^Home:	When the control key is held down as the Home key is pressed, the cursor will be moved to the beginning of the first line of the file.
End:	When the End key is pressed once, the cursor will move to the end of the current line (positioned just before the CRLF end of line sequence). If the End key is pressed twice in succession, the cursor will move to the beginning of the last line on the screen.
^End:	With the control key held down, the End key will move the cursor to the end of the file.
PgUp:	Moves up twenty lines of text and redisplays the screen.
^PgUp:	Scrolls the screen up one line without moving the cursor.
PgDn:	Moves down twenty lines of text and redisplays the screen.
^PgDn:	Scrolls the screen down one line without moving the cursor.
UpArrow:	The up-arrow key moves the cursor up one line. The column that the cursor is in remains the same. If the cursor is positioned beyond the end of a line because of this action, the visible cursor is shown beyond the end of the line but is logically located just before the CRLF

	sequence (The cursor is moved to this location when some other operation is performed.) If the cursor is already on the top line of the screen, the screen is scrolled down one line to show the new line.
DownArrow:	The down-arrow key moves the cursor down one line. Again, the visible cursor remains in the same column as described above. If the cursor is already on the last line of the screen, the screen is scrolled up one line to show the new line.
LeftArrow:	The left-arrow key moves the cursor one character to the left. If the cursor is at the left edge of the screen, and the screen has been scrolled to the right, the screen will scroll back to the left by 15 character locations to show the new cursor position. If the screen had not been scrolled implying that the cursor was on the first character of the line, the cursor moves to the end of the previous line.
^LeftArrow:	With the control key held down, the cursor will move to the left in word increments rather than character increments. Each time this combination is pressed, the cursor will move to the first character of the previous word where word is defined as a sequence of letters or digits. Any other character separates the words.
RightArrow:	The right-arrow key moves the cursor one character to the right. If the cursor is at the right edge of the screen and more text exists in the current line, the screen is scrolled to the right by 15 characters to show the new location. If the cursor was positioned at the end of the line, then the cursor is moved to the beginning of the next line.
^RightArrow:	This combination moves the cursor to the beginning of the next word.
Return:	The return key is normally used to insert a CRLF end of line sequence into the text, thereby positioning the cursor at the beginning of the next line. If the return key is pressed while in command mode, the cursor will simply move to the beginning of the next line.

Editing Keys

Backspace:	The backspace key deletes the character to the left of the cursor. If the cursor is positioned at the beginning of a line, the CRLF sequence is removed and the two lines are joined to form a single line.

Del:	The delete key deletes the character at the cursor. If the cursor is positioned on the CRLF end of line sequence, then the next line is joined with the current line.
Ins:	The Ins key is used to toggle between Insert and Exchange modes. At the command level, it will place the editor into Insert mode.
F1-F8:	The function keys F1 through F8 are available for user-defined macros. Macros may be saved with the Macro-Save command.
F10:	If you are using a split-screen to display two files concurrently, the F10 key will temporarily expand the current file display to fill the entire screen. Switching to the other file will reset to the split-screen.
^C or ^Break:	Holding down the control (Ctrl) key and hitting the letter 'C' or the Break key (Scroll Lock) will normally stop the execution of a command (where reasonable). This is useful when you decide not to execute the Find command and are in the middle of typing in the search string. Typing control-C will abort the Find command without modifying the old search string. This key combination will also stop an executing macro.
^Return:	Deletes from the cursor to the end of the line. This command may be used to edit the command line (e.g., Find, Replace, Initialize).

The DOS Key

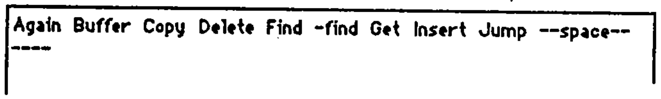
Under MS-DOS 2.0 and later versions of the operating system, the F9 function key allows another command shell to be executed while the editor and text remain in memory. When the F9 key is pressed, the screen will display the DOS copyright message and will prompt for a command. You can execute any command, even another copy of the editor (although this is not recommended because of conflicts with the spill files). When you want to return to the editor, type the DOS command

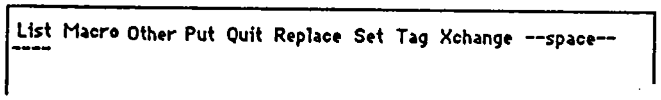
```
exit
```

and the text will be redisplayed as if the F9 key never had been pressed. DOS, SEE, and your text occupy about 128K. You must have at least an additional 64K of unused memory in your machine to use the DOS feature.

Commands

In command mode, the menu line displays the commands available for editing and manipulating the text. Since the names of the commands are too long for a single menu line, the menu is broken into two parts. To toggle between each part of the command menu, press the space bar.


Again Buffer Copy Delete Find -find Get Insert Jump --space--
----



Command Menus

To invoke a command, type the first letter of the command. To terminate a command, press the escape <Esc> key. A command may be aborted by holding down the control key, Ctrl, and typing the letter C (control-C).

Many commands will take a repetition count to execute the command multiple times before completing. The repetition count takes the form of a decimal number or a slash (indicating a very large number). It is entered prior to typing the first letter of the command. Some commands — Find, -find, and Replace — may be given a question mark (?) repetition count indicating that the editor should prompt after each string is found. Note that at the command level, the cursor movement keys may also be repeated by using a repetition count. This also means that if a mistake is made in the repetition count, the Backspace key cannot be used to correct the mistake. The command must be aborted .

In the following descriptions of the commands, <rep> indicates that the command takes a repetition count and <rep | ?> indicates that it will take a repetition count or question mark repetition count.

Commands that need further information (i.e., Find) interact with the user on the second line of the display — the message line. Each command that interacts on the message line displays the current value of the information sought. You can use the current value by pressing <RETURN>, or you can use all the editing facilities of SEE to edit the current value before pressing <RETURN>.

<rep> Again

The Again command repeats the action of the last Find, -find, or replace command without any prompting. For example, if a Find command is executed to locate the string "hello", then executing the Again command will find the next occurrence of the string "hello".

Buffer

The Buffer command is used to copy a block of text into the copy buffer. The copy buffer is an internal buffer used to hold the last buffered or deleted (with the Delete command) item. To use the Buffer command, move the cursor to the beginning of the block to be buffered and type 'B' for Buffer. The character at the cursor will be temporarily overwritten with a block to indicate the beginning of the block. The menu line will be replaced with the new menu line:

```
    Buffer:  <cursor keys> <esc | B>  Again  Find  -find  Jump
```

Now move the cursor to the end of the block, either with the cursor movement keys or with the Again, Find, -find and Jump commands. These commands may be preceded with a repetition count. When the cursor is positioned at the end of the block, press the Esc key or the letter 'B' to terminate the buffering operation. SEE will copy the contents of the block into its copy buffer. The previous contents of the copy buffer are thrown away.

<rep> Copy

The copy command inserts the contents of the copy buffer at the current cursor location. If a repetition count is given, the contents of the buffer will be inserted that many times.

Delete

The delete command is used to delete a block of text. The deleted text is placed in the copy buffer, as mentioned in the Buffer command. To use the Delete command, first move the cursor to the beginning of the block of text to be deleted and type 'D' for Delete. The character under the cursor will be temporarily overwritten with a block to indicate the beginning of the block. The menu line will be replaced with the new menu line:

```
    Delete:  <cursor keys> <esc | D>  Again  Find  -find  Jump
```

Now move the cursor to the end of the block, either with the cursor movement keys or with the Again, Find, -find, and Jump commands. These commands may be preceded with a repetition count. When the cursor is positioned at the end of the block, press the Esc key or the letter 'D' to delete the block. The text will be removed and placed in the copy buffer.

<rep | ?> Find

The find command is used to locate the next occurrence of a given string. The search runs from the cursor location to the end of the file. To use the Find command, type the letter 'F' for Find. The Find command will then prompt for the search string. The last string given in a Find, -find, List, or Replace command is displayed on the message line. If the same string is to be found, hit the Esc or Return key to select the argument. Otherwise, edit the message line to create the new search string. When the string is entered, press the backquote or Return key to indicate completion. The Find command will then search for the next matching string. If found, the cursor will be moved to the character following the string, otherwise the message:

```
    can't find "<string>"
```

will be displayed and the cursor will not move. <string> is the search string. If a repetition count is given, the string will be located that many times before the command is done. For example, typing the command stream:

    3 F hello <Return>

will place the cursor after the third occurrence of the string "hello". If a question mark (?) is given as the repetition count, the editor will move the cursor to the next occurrence of the string and prompt with the message:

```
    continue? (y/n)
```

Typing the letter 'Y' will move the cursor to the next occurrence of the search string. Any other character will stop the Find command.

<rep | ?> -find

The -find command works similarly to the Find command except that the text is searched backwards from the cursor to the beginning of the file. When the -find command terminates, the cursor is left on the character prior to the located string. The question mark repetition count works as in the Find command.

Get

The Get command is used to insert the contents of a file into the current file. The text from the file is inserted at the cursor location. To use the Get command, position the cursor at the insertion point and type the letter 'G'. The Get command will prompt for a filename. Enter the filename and type <Return>. The Get command will prompt with

```
    reading from <filename> ...
```

and attempt to read and insert the text from the file. If everything goes well, the word "completed" will be added to the prompt. If an error occurs (usually meaning that the file does not exist), the words "can't read file" will be appended to the prompt. Finally, if the buffer was filled as a result of the Get command, the words "buffer filled" will be appended to the prompt indicating that only part of the file was inserted.

Insert

The Insert command is used to place the editor into insert mode. Once in insert mode, characters other than the command characters will be inserted in the text at the cursor location. The cursor movement characters always move the cursor appropriately. The Insert command does not normally do anything with the repetition count. However, if the '/' repetition character is specified, then a newline character is inserted at the cursor location before entering insert mode. If the Ins key is pressed, the mode will be changed to Exchange mode. To terminate the insert mode, type the Esc character.

<rep> Jump

The Jump command is used to move to a location previously marked with the Tag command or for moving to a line when given a line number. When a repetition count is given, the cursor will be moved to the beginning of the corresponding line (the line number given by the repetition count). Otherwise, the Jump command will display the menu:

```
Jump:  A  B  C  D
```

indicating the four tag names to use. If one of these letters is typed, the cursor will be moved to the location associated with the tag. This location is set with the Tag command. If the tag has not been set, the cursor will move to the end of the file.

<u><rep> List</u>

The List command is used to display all lines containing the given string. The List command prompts for the search string the same way as the Find command. Once the search string has been entered, the List command temporarily takes over the screen and displays all lines, beginning from the cursor location, which contain the search string. A line will be listed once, even if it contains multiple instances of the search string. After the screen is filled with lines, the prompt will read:

```
hit a key to continue
```

Any key other than control-C will display the next set of lines. If there are no more lines with matching strings, the screen reverts to its normal display with the cursor positioned after the last matching string. If no repetition count is given, all occurences are assumed. Otherwise, the repetition count will control the number of times the List command will search for the string.

<u>Macro</u>

The Macro command is used to record input from the keyboard and the mouse. This recording can then be played back to perform the same sequence of operations beginning at another point in the text. Thus macros give the ability of creating custom functions built from the standard set of operations. There are eight definable macro keys, F1 through F8. When one of these function keys is pressed, the macro associated with the key is replayed. Note that macro keys and the Macro command <u>cannot</u> be recorded.

When the Macro command is invoked, the menu:

```
    Macro:  Delete  Load  Record  Save
```

will appear with the following meanings:

`Delete`: used to delete a macro definition. Delete prompts with the menu line:

```
        select function key:  F1 - F8
```

When a function key is selected, the macro associated with the key will be removed. Typing the Esc key will exit the command without deleting any macro.

`Load`: used to reload the macros and controls settings from the "see.mac" file. This file is created by the Save command.

`Record`: used to start the macro recording. Record prompts for the macro number with the menu:

```
        select function key: F1 - F8
```

When the function key is selected, the old macro associated with that key, if any, is deleted and a new recording is begun. All input will be recorded as part of the macro. To terminate the recording, reinvoke the Macro command by typing the letter 'M'. Now when the command key is held down and the number is typed, the recording will be replayed as if the inputs were coming from the keyboard.

`Save`: used to save the macro definitions and control settings (see the Set command) into the file named "see.mac" in the current directory. This file is read, if it exists in the current directory, when the editor is first invoked and when the Load command is used. If the file does not exist in the current directory, then each directory in the PATH system parameter is searched.

For example, the following sequence of commands will create macro F1 which can be used to delete the current line:

```
    M  R  <F1>  <Home>  D  <down-arrow>  D  M
```

Now when the F1 key is typed, the line that the cursor is on will be deleted.

Other

The Other command is used to toggle between the two files available for editing. The first time the Other command is used, it will prompt for a command line as in the Quit-Initialize command. Subsequent uses of the Other command will change the active file from one to the other. If the F10 key had been used to temporarily expand a split-screen display to use the entire screen, use of the Other command will reset the display to the split-screen configuration.

Put

The Put command is used to write a block of text out to a separate file. To use the Put command, move the cursor to the beginning of the block to be written and type the letter 'P'. The character under the cursor will be temporarily overwritten with a block character to indicate the beginning of the block. The menu line will be replaced with the new menu line:

```
Put:  <cursor keys> <esc | P>  Again  Find  -find  Jump
```

Now move the cursor to the end of the block, either with the cursor keys or with the Again, Find, -find, and Jump commands. These commands may be preceded with a repetition count. When the cursor is positioned at the end of the block, press Esc or the letter 'P' to select the end of the block. The Put command will then prompt for a filename. Enter the filename and type <Return>; the block of text will be written to the file.

Quit

The Quit command is used to terminate an editing session. When the letter 'Q' is typed, the Quit command will display the menu:

```
Quit:  BAKup  Compile Exit  Initialize  Save-exit Update  Write
```

and will show the name of the file, an indication if the memory buffer has been modified, and the size of the file, on the message line. To leave the Quit menu without executing any commands, type the Esc character. The menu items have the following meanings:

`BAKup`: causes SEE to change the extension of the old file to .BAK and then write the contents of the memory buffer to the filename given on the invocation line. If a new file is being edited, no .BAK file is created.

`Compile:` causes the SEE editor to invoke the C88 compiler using the C88 Flags from the SET menu. The message **compiling** is displayed on the message line. If an errors occurs, the error message is displayed on the message line, and SEE resumes editing the file at the error line.

If no error occurs, SEE will invoke BIND using the BIND Flag from the SET menu. The message **binding** is displayed on the message line. If an error occurs, the error message is displayed on the message line.

The file is not saved prior to the compilation.

`Exit:` causes the SEE editor to exit back to the system. If the memory copy of the file has been modified, SEE will prompt with the question:

```
ignore changes? (y/n)
```

Typing 'y' will leave the editor and the changes made to the memory image of the file will be lost. Any other character will abort the Exit command.

`Initialize:` causes the SEE editor to reinitialize the editor and prompt for a new invocation line (excluding the SEE program name). If the text has been modified and not saved, SEE will prompt as if Exit had been selected, giving one last chance to save the changes to the file. The new file is then read in and the editor is restarted. Note that the macros and the copy buffer are left intact and can be used with the new file.

`Save-exit:` writes out the file to the disk and exits from the editor without further prompting.

`Update:` writes a copy of the memory buffer out to the file given on the invocation line. This command is useful for quickly saving the contents of the memory buffer out to the disk to prevent a large loss of data if a fatal error should occur (either software or hardware).

`Write:` writes a copy of the text to a specified file. The Write command will prompt for filename and will then write the text to that file. This command is usually used when no filename was given on the invocation line.

<u><rep | ?> Replace</u>

The Replace command is used to locate a specific string of characters and replace it with another string. Replace uses the same search string specified in the Find, -find, and List commands. To replace a string, type the letter 'R' and enter the search string (or just type Return if the current search string is correct). Then enter the replacement string and type <Return>. The editor will find the <u>next</u> occurrence of the search string and replace it with the replacement string. If the search string cannot be found, the following message will be displayed:

```
    cannot find "<search string>"
```

The repetition count controls the number of times the replacement will be performed. To replace all occurrences, move the cursor to the beginning of the file and use '/' for the repetition count. If the question mark (?) is given as the repetition count, then before the string is replaced, the editor will prompt with:

```
    replace? (y/n) or quit (q)
```

Typing the letter 'Y' will replace the string and the cursor will move to the next occurrence of the search string. Typing the letter 'N' will simply move the cursor to the next occurrence of the search string. And typing the letter 'Q' will abort the Replace command.

<u>Set</u>

The Set command is used to change several controls in SEE; tab width, indentation, case sensitivity on search strings and a special auto-insert mode. The values of the controls may be saved with the Macro Save command so that the settings will be the same each time the editor is invoked. The Set command will display one of two menus:

```
Set:Auto-ins(off) Case(no) Flags Height(0) Indent(yes) PC -space-

Set:Right(80) Spill(D) Tabs(8) Word-wrap(off) '{'-indent(2) -space-
```

The current settings of the controls are displayed in parentheses. To change a control, pick its menu item and follow the prompts. The controls are defined as follows:

`Auto-ins:` This control forces the editor into insert mode after each command. To execute a single command, type the Esc key to temporarily terminate the insert mode and bring up the command menu. Select a command as usual. After the command executes, SEE will automatically place itself back in Insert mode. Selecting this menu item will display one of the following two messages, depending on the state of the control:

if the Auto-insert control is off (default)

```
Set auto-insert mode? (y/n)
```

otherwise

```
Reset auto-insert mode? (y/n)
```

Typing 'Y' will change the control from one state to the other. Anything else will leave it alone.

`Case:` This control is used while searching for strings in the Find, -find, List, and Replace commands. When the control is on, the case of the search string and the text is ignored during the string comparison, so the string "AbC" is equal to the string "aBc". When this control is off, the case of the characters in the string must match exactly. Depending on the state of the case-ignore flag, one of the following messages will be displayed when this menu item is selected:

if the case-ignore control is on (default)

```
Make case significant on searches? (y/n)
```

otherwise

```
Ignore case of searches? (y/n)
```

Typing 'Y' will change the state of the control.

`Flags:` This control specifies the command line for invoking the compiler (C88) and the linker (BIND). The choices for this menu are:

```
Set Flags: Bind C88
```

The Bind option displays the line that will be use to invoke BIND. The default is:

```
    BIND %0
```

BIND is the name of the linker. *%0* contains the name of the file being edited. Edit the flag to add any BIND options you wish.

If you are compiling several modules, set the Bind Flag to a NULL string (use ^RETURN), so that BIND is not called for each module. Then, use the F9 key to invoke DOS and BIND the files manually.

The C88 Flag displays the line that will be use to invoke C88. The default is:

```
    C88 %0
```

C88 is the name of the compiler. *%0* contains the name of the file being edited. Edit the flag to add any C88 options you wish.

`Height:` This control controls how two files are displayed on the screen. The submenu is:

```
    enter second screen height (0, 8..17)
```

If 0 is entered, the second file is shown alone on its own screen. Using the Other command flips the display between the two displays. This is the default height.

If a number between 8 and 17 is entered, both files are displayed together. A dashed line separates the two areas. The height of the second area is the number of lines entered.

`Indent:` This control indicates whether the blanks and tabs from the previous line are copied to the beginning of the the new line when a Return is inserted. When this control is on, the indentation is copied. This provides an aligned left margin to the indented text. When this control is off, no indentation is copied when a Return is entered and the cursor moves to the left edge of the screen. Depending on the state of the Indent control, selecting the Indent menu item will result in one of the following messages:

if the Indent control is on (default)

```
    Reset auto-indent mode? (y/n)
```

otherwise

```
    Set auto-insert mode? (y/n)
```

Typing 'Y' will change the state of the control.

`PC:` This menu item selects the IBM-PC specific information. These settings may be valid for other direct clones but it is not guaranteed. The following menu will be displayed:

```
Set PC:Add-^Z Cursor-height  Foreground-color  Background-color
```

`Add-^Z:` Controls whether a DOS 1.0 EOF character (CTRL-Z, 0x1A) is appended to the end of the file. Selecting this menu will display one of the following two messages, depending upon the state of the control:

If the ^Z control is off (default)

```
    add control-Z at the end of files? (y/n)
```

otherwise

```
    stop adding control-Z? (y/n)
```

Typing 'Y' will change the state of the control.

`Cursor-height:` Sets the height, in pixels, of the character cell size. By enabling this control, the cursor will change shapes according to the mode that the editor is in; a double bar for command mode, a single bar for insert mode and a block for exchange mode. Enter

0	disable this feature,
8	color graphics adapter,
12	monochrome adapter.

`Foreground-color`: Sets the foreground color attribute. The colors are defined by the IBM-PC as follows:

0	black,
1	blue,
2	green,
3	cyan,
4	red,
5	magenta,
6	brown,
7	light grey,
8	dark grey,
9	light blue,
10	light green,
11	light cyan,
12	light red,
13	light magenta,
14	yellow,
15	white.

`Background-color`: Sets the background color attribute. The background colors for the IBM-PC are defined as follows:

0	black,
1	blue,
2	green,
3	cyan,
4	red,
5	magenta,
6	brown,
7	light grey.

Values above 7 cause the characters to blink.

`Right`: This control sets the character column for the Wrap command and the automatic word-wrap mode. Words which extend beyond this column are moved to the next line. The following message is displayed:

```
enter wrap right column:
```

Enter a number between 0 and 255. An invalid specification is signaled with

```
bad wrap width (0 < x < 256)
```

Spill: This control determines the drive on which the editor's spill files will be created. A '@' indicates the current default drive. The following message will be displayed:

```
enter spill device letter: (A-Z, 0 for default)
```

Type a single letter to signify which drive to use or the '0' character to indicate the use of the current default drive.

If spill files have already been opened, they will be moved to the new drive (the contents of the copy buffer will also be deleted). This is useful if the original spill disk becomes full and another disk is available.

Tabs: This control determines the expansion factor of tab characters in the text. The following message will be displayed:

```
select tab size (1..19)
```

By default, this value is 4. However, if the file on the invocation line has an extension which starts with the letter 'A' (as in xxx.a), then the tab size will be set to eight; a useful size when writing in assembly language. If the extension starts with the letter 'C', then the tab size is set to four. Otherwise the tab size remains at its current setting. The tab size may be a value from one to nineteen indicating that the tab stop locations will be separated by one to nineteen character locations, respectively.

Word-wrap: When this control is on, the editor will automatically move words to the next line if the current column is greater than the right column Selecting this menu will display one of the following two messages, depending upon the state of the control:

If the ^Z control is off (default)

```
set word-wrap mode? (y/n)
```

otherwise

```
reset word-wrap mode? (y/n)
```

Typing 'Y' will change the state of the control.

'{'-indent: This is a special indentation mode for assisting in C programming. The following mesaage will be displayed:

```
enter indent mode (0, 1, 2)
```

An error is signaled by

```
bad indent mode
```

When this control is on and the Indent control is on, the editor will automatically add an extra tab character to the indentation when a <Return> is inserted just after the left brace ({) character. There are two possibilities for <Return>s which follow the right brace (}) character. If mode 1 is selected and a tab character preceded the right brace character, it will be removed and the indent level reduced accordingly. This corresponds to the following type of indentation:

```
main()
{
      int i;
      for (i = 1; i < 10; i++)
      {
            printf("hello, world\n");
      }
}
```

If mode 2 is selected, then the indentation of the new line is decreased by a tab if the Return was inserted just after the right-brace (}) character. This corresponds to the following type of indentation:

```
main() {
      int i;
      for (i = 1; i < 10; i++) {
            printf("hello, world\n");
            }
      }
```

Tag

The Tag command is used to set markers in the text file. Once a tag is set, the marked character can be located with the Jump command regardless of the insertions and deletions around the marked character (unless the marked character is deleted). The Tag command displays the menu:

```
    Tag:  A  B  C  D
```

where A, B, C, and D correspond to the four tags available. To use the Tag command, move the cursor to the character to be marked and type 'T'. Now select one of the tag names by typing the corresponding letter.

Version/View

When the cursor is at the begining of the file, this command displays the SEE version number on the second line of the display. With the cursor at any other location in the file, it redisplays the current screen with the line containing the cursor at the third line of the display.

Wrap

The Wrap command is used to reformat a paragraph. All of the lines, starting with the line that the cursor is currently on to the next blank line, are reformatted to make sure no word extends beyond the right margin (set by the Right-col control). Indentation for the lines is determined by the indentation of the first line of the paragraph. The Wrap command requires a confirmation to avoid wrapping code by mistake. The letter 'W' may also be used to confirm the Wrap operation.

Xchange

This mode is similar to Insert mode except that characters in the text are overwritten by the new characters. The only characters not overwritten are Returns. An attempt to overwrite a Return simply inserts the character prior to the Return. If the Ins key is pressed while in Exchange mode, the mode will be changed to Insert mode.

Finally, there are a few single character commands which are not listed on the menu line but may be of use:

#

The number sign (#) command displays the current line number on the message line.

{ } ( ) [ ]

When the cursor is on a left brace '{', left parenthesis '(' or left bracket '[' and one of these command characters is typed, the cursor will be moved forward to the corresponding right brace '}', right parenthesis ')', or right bracket ']'. If the cursor is on a right character '},),]', it will move backward to the corresponding left character '{,(,['. Note that this command does not know about comments so unmatched characters will confuse the search routine.

<rep> \

The backslash command is used to insert literal characters into the text by entering their decimal equivalents. When backslash is typed, the editor will prompt for a decimal value. Numbers from 0 to 255 are valid but 254 and 255 have special meaning to SEE. The repetition count determines the number of times the command will prompt for input.

Configuration

Distributed with the package, are a number of files used to reconfigure the editor to run on other DOS based machines with different keyboards and/or screens:

SEE.O:	relocatable object file
PCIO.A:	source code for an IBM-PC BIOS based interface.
CONFIG.C:	source code for terminal based screen interfaces. Contains interfaces for ANSI terminals, a Hazeltine 1500, Dec VT-52 and the Zenith Z100.

The PCIO.A and CONFIG.C files should contain enough information in the comments to build your own interface if necessary.

To build the editor, compile/assemble one of the interface files, or one of your own making, and link it with the editor with the following bind command:

```
bind see config
```

This will generate a new SEE.EXE file with your interfaces linked in instead of the standard IBM-PC interfaces.

Chapter 4

The C88 C Compiler

Introduction	4.1
Invocation	4.1
Examples	4.3
The C Language	
Environment	
Character Set	4.4
Trigraph Sequences	4.4
Language	
Keywords	4.5
Identifiers	4.5
Floating constants	4.5
Integer constants	4.6
Character constants	4.6
String constants	4.6
Hardware data types	4.7
Enumerated type	4.8
Function prototyping	4.8
Preprocessor	
Conditional compilation	4.9
Source file inclusion	4.10
Macro replacement	4.11
Line control	4.11
Error	4.11
Pragma	4.11
Null	4.12
Predefined macros	4.12
Extensions	
Asm	4.12
Case range	4.13
Restrictions	
Forward references	4.14
Externs	4.14
Large Case Option	4.15

Introduction

C88 is the C compiler for the 8088/8086 family of microprocessors. It accepts C source programs as input and produces object files. The compiler supports both the Small memory model which efficiently utilizes the 8088/8086 architecture but limits a program to 64KB code and 64KB of data, and the Large Memory Model which is limited only by the amount of available memory.

Invocation

C88 <filename> [options]

<filename> is the name of the file containing the C source. If it does not include an extension, the extension '.C' is assumed.

You may use a hyphen, '-', to enter the source from the predefined file `stdin`. The file must be a disk file, compiling from the keyboard is not supported. You can redirect the input

```
c88 - <filename
```

or you can use a pipe

```
yourpgm | c88 -
```

Options: The case of the option is not significant. Each option should be separated from other options by blanks. Options may be preceded with the dash (-) character.

A - assembly output. This option indicates that the compiler should produce an assembly language source file instead of an object file. The name of the assembly language file will be the same as the name of the source file but will have the extension '.A'.

B - big. This option instructs the compiler to produce Large Case Memory Model output. You need the Large Case Option (not included with the DeSmet C Development Package) to use this option.

C - produce check information. This option causes the compiler to generate information for BIND to create the .CHK file used by the debugger and profiler.

D<name> - compiler drive specification. The compiler assumes that the files GEN.EXE and ASM88.EXE are in the default directory on the current drive. This option is used to inform the compiler that the files are on a different drive. For example, if the compiler is on drive 'M', then the option 'DM' is needed.

Under MS-DOS 2.0 and later versions of the operating system, this option is rarely needed as the system PATH variable is also used to find the other passes of the compiler.

E - run the preprocessor and output the result to the predefined file `stdout`. All macros and include files are resolved.

I<name> - include path name. This option overrides the default drive/directory for files included with the #include control. The directory name must end with a trailing backslash (\) character (e.g. `-ic:\src\include\`). See the Preprocessor section for `#include` details.

M This option is used to produce Intel object files rather than the standard .O object file format. To work properly, the file TOOBJ.EXE must be in a directory in the PATH.

N<defname>=<defvalue> - specify `#define` name and value. Used to set debugging switches or constant values without editing the file. This option is equivalent to

```
    #define defname defvalue
```

at the beginning of the program. To set <defname> to 1, enter

n<defname>, which is equivalent to

```
    #define defname 1
```

Spaces are not allowed.

O<filename> - output filename. The compiler will produce an object file with the specified name. If the name lacks an extension, the extension '.O' will be added. The default object name is the same as the source name with the extension of '.O'.

P<switch> - sets the indicated pragma switch on. The switches are:

T - requests that ANSI trigraph sequences be processsed.

W - requests the display of all warning messages. Structure assignment, structure arguments, and prototype conversion warning messages are not normally displayed.

X - requests that the extended keywords be recognized. Use this switch when you use the `#asm` feature to embed assembler into your C program.

T<drive> This option specifies the drive that the compiler should use for its temporary files. If not specified, the compiler will build its temporary files on the default drive. If this drive is close to being full, the 'T' option should be used to change the drive for the compiler work files. Also, if the RAM Disk has been installed, placing the temporary files there will drastically cut the time needed to compile a program.

Examples

```
C88 blip
```

compiles the file named blip.c. The object file will be named blip.o.

```
C88 blip b
```

compiles the file named blip.c using the Large Case Option. The object file will be named blip.o. You must use BBIND to link this program.

```
m:C88 b:blip.ccc tm dm
```

runs the compiler from drive M on the file b:blip.ccc. Temporary files are also written on drive M. Note the use of the D option to indicate the location of the other passes of the compiler. The object file will also be named blip.o.

```
c88 blip pwx
```

compiles blip.c in the current directory. Structure assignment, structure argument, and prototype coersion warnings are reported. The extended keywords are recognized.

The C Language

DC88 compiles C programs that conform to the definition of the C language as described in the Draft Proposed American National Standard for Information Systems — Programming Language C.

ENVIRONMENT

Character Set

DC88 recognizes the following characters:

letters	the 52 upper-case and lower-case letters of the English alphabet and the underscore [`a-z,A-Z,_`]
digits	the ten decimal digits [`0-9`]
white-space	the space, horizontal tab, vertical tab, form feed, carriage return, and line feed characters. Comments, sequences of characters begun with a `/*` sequence and ending with a `*/` sequence, are equivalent to a space character. Comments do **not** nest.

others	the following 28 graphics charcters
	`! " # % & ' ( ) * + , - . /`
	`: ; < = > ? [\ ] ^ {
		} ~`

In addition, any of the 255 characters in a byte (excluding 0) are valid in a string constant.

Trigraph sequences

When trigraph processing is enabled (through the **-pt** command line switch or the `#pragma trigraph` option), the following sequences of three characters are replaced with the corresponding single character.

```
??=  →  #          ??(  →  [          ??/  →  \
??)  →  ]          ??'  →  ^          ??<  →  {
??!  →  |          ??>  →  }          ??-  →  ~
```

LANGUAGE

Keywords

The following tokens are reserved as keywords of the language. A * indicates an extended keyword (enabled through the **-px** command line switch or the `#pragma extended` option).

* `asm`	`double`	`if`	`sizeof`
`auto`	`else`	`int`	`static`
`break`	`enum`	* `interrupt`	`struct`
`case`	`extern`	`long`	`switch`
* `cdecl`	* `far`	* `near`	`typedef`
`char`	`float`	* `pascal`	`union`
`const`	`for`	`register`	`unsigned`
`continue`	* `fortran`	`return`	`void`
`default`	`goto`	`short`	`volatile`
`do`	* `huge`	`signed`	`while`

Identifiers

An identifier is a sequence of letters (which include the underscore) and digits. The first character must be a letter or an underscore. Only the first 31 characters are significant. Corresponding lower-case and upper-case letters are different.

Floating constants

A floating constant has a value part, followed by an optional exponent part, followed by an optional suffix that specifies its type. The value part consists of an optional digit sequence representing the whole-number part, followed by a period (.), followed by a digit sequence representing the fraction part. The exponent part consists of either an `e` or an `E`, followed by an optionally signed digit sequence. Either the whole-number part or the fraction part must be present; either the period or the exponent part must be present.

The digit sequences are interpreted as decimal integers. The exponent indicates the power of 10 by which the value part is to be scaled.

An unsuffixed floating constant has type `double`. If suffixed by the letter `f` or `F`, it has type `float`. If suffixed by the letter `l` or `L`, it has type `double`.

Integer constants

An integer constant begins with a digit, but has no period or exponent part. It may have a prefix that specifies its base and a suffix that specifies its type.

A decimal constant begins with a nonzero digit and consists of a sequence of decimal digits. An octal constant consists of the prefix `0` optionally followed by a sequence of the digits `0` through `7` only. A hexadecimal constant consists of the prefix `0x` or `0X` followed by a sequence of decimal digits and the letters `a` (or `A`) through `f` (or `F`) with values 10 through 15 respectively.

An unsuffixed integer constant has type `int` or `long` depending on its size. If suffixed by the letter `u` or `U`, it has type `unsigned`. If suffixed by the letter `l` or `L`, it has type `long`.

Character constants

A character constant is a sequence of one or more characters enclosed in single-quotes, as in `'x'` or `'ab'`. The characters can be any of the 255 byte values, excluding 0. The single-quote ', the double-quote ", the question-mark ?, the backslash \, and arbitrary integer values, are representable according to the following table of escape sequences.

`\'`	`0x2C`	single-quote '	`\"`	`0x22`	double-quote "
`\?`	`0x3F`	question-mark ?	`\\`	`0x5C`	backslash \
`\ooo`	1 to 3 digit octal value		`\xhhh`	1 to 3 digit hexadecimal value	
`\a`	`0x07`	alarm	`\b`	`0x08`	backspace
`\f`	`0x0C`	formfeed	`\n`	`0x0A`	linefeed
`\r`	`0x0D`	carriage return	`\t`	`0x09`	horizontal tab
`\v`	`0x0B`	vertical tab			

String Constants

A string constant is a sequence of zero or more characters enclosed in double-quotes, as in "xyz". Each character in the sequence is treated as if it were in a character constant, except that the single-quote ' can be represented either by itself or the escape sequence `\'`. The double-quote " must be represented by the escape sequence `\"`.

The maximum size for a source file string constant is 255 bytes. Larger object file string constants can be constructed by concatenating source file string constants, as in:

```
    char xyz[] = "aaaaaaaaaaaa"
                 "bbbbbbbbbbbb"
                 "cccccccccccc";
```

which is equivalent to

```
    char xyz[] = "aaaaaaaaaaaabbbbbbbbbbbbcccccccccccc";
```

Hardware data types

DC88 Type	8088 Type	Description
char	BYTE	Unsigned byte with a range of 0 to 255.
int **short**	WORD	Signed integer with a range of -32768 to 32767.
unsigned	WORD	Unsigned integer with a range of 0 to 65535.
long	DWORD	Signed integer with a range of -2147483648 to 2147483647.
float	DWORD	Four byte IEEE floating point value. A `float` has about 7 digits of precision and has a range of about 1.E-36 to1.E+36.
double	QWORD	Eight byte IEEE floating point value. A `double` has about 13 digits of precision and a range of about 1.E-303 to 1.E+303.
(pointer)	WORD	In the Small Case Memory Model, pointers are two bytes, limiting total data space to 64KB.
	DWORD	In the Large Case Memory Model, pointers are four bytes.

To take advantage of the 8088/8086 instruction set, expressions involving only *char* types are not coerced to int before evaluation. The sum of a char equal to 255 and a char equal to 1 is 0 rather than 256. Constants are considered to be int values so that constant plus char is a two byte integer operation.

Enumerated type

Enumerated type provides a convenient method of declaring an ordered set of named constants. Values start with zero, and may be reassigned with a `name = value` expression. The same value may be assigned to several names. For example

```
enum color {red, blue=4, green} ca, *cp;
enum color cb;
if(ca == red)
   cb = *cp = green;
```

is equivalent to

```
#define red   0
#define blue  4
#define green 5
int ca, *cp;
int cb;

if(ca == red)
   cb = *cp = green;
```

Function prototyping

Both the type of value returned by a function, and the type(s) of the arguments to the function may be specified. To specify the type of each argument to a function, place the types of the argument, separated by commas, between the parentheses in the function declaration. For example to indicate that a function takes a `double`:

```
double sqrt(double);
```

If you call `sqrt(1)`, the integer 1 will be converted to a `double`. If you want to know when these conversions are made, use the **-pw** command line option, or specify `#pragma warning` in your program.

To indicate that a function takes no arguments, specify:

```
void abort(void);
```

To indicate that an unknown number of arguments, of unknown type, are valid, specify:

```
int printf(char *, ...);
```

`int xxx();` is equivalent to `int xxx(...);`

You can also define functions using the prototypical style, as in

```
int main(int argc, char *argv[]) {
```

Only prototypically declared functions are checked for possible argument conversions.

PREPROCESSOR

Before any interpretation of source file text is started, all occurrences of a backslash-newline sequences are deleted. Thus you can break text anywhere for cosmetic purposes without changing its meaning, as in

```
#defi\
ne FO\
O 10\
24
```

which is equivalent to `#define FOO 1024`

A preprocessing directive consists of a sequence of preprocessing tokens begun by a # character that is the first character of a line (following optional whitespace characters), and ended by a linefeed character.

Conditional compilation

`#if expr, #elif expr, #ifdef id, #ifndef id, #else, #endif`

`#if` and `#elif` evaluate *expr*. *expr* must be a constant expression, but may not contain a `sizeof` operator, a cast, or an enumeration constant. *expr* may contain a unary expression of the form

`defined` *identifier*

or

`defined (` *identifier* `)`

which evauates to 1 if *identifier* is currently defined as a macro name, 0 otherwise. All constants are evaluated as `long`.

`#ifdef` and `#ifndef` are equivalent to `#if defined id` and `#if !defined id`, respectively.

If the test evaluates to 0, the source code up to the next `#elif`, `#else`, or `#endif` is skipped. If the next directive is `#elif`, its *expr* is evaluated. If it is `#else`, the following source code, up to the terminating `#endif`, is passed to the compiler. If it is `#endif`, the conditional compilation is complete.

If the test evaluates to non-zero, the source code up to the next `#elif`, `#else`, or `#endif` is passed to the compiler. Then the remaining source code, up to the terminating `#endif`, is skipped.

Conditional compilation groups can be nested.

Source file inclusion

```
#include name
```

Includes other files into the program. #include's can be nested to a maximum depth of 20.

`#include "filename"` will search the default directory for the file *filename* .

`#include <filename>` will first search the default directory for *filename* . If the file was not found, the environment (see the DOS SET command) is searched for the key `DSINC`. If `DSINC` is not found, the environment is searched for the key `INCLUDE` . If `INCLUDE` is not found, a "cannot open include file" error is generated. The `DSINC` key is used in case you have other applications that also use the key `INCLUDE`.

If the key is found, it should contain a set of directory prefixes separated by semi-colons and terminated with a backslash (\ ). For example, if `DSINC` is set as follows

```
       C>set DSINC=c:\;c:\usr\include\
```

then the line `#include <world.h>` would cause C88 to search for

```
       world.h
       c:\world.h
       c:\usr\include\world.h
```

`#include name` behaves like the previous two examples, where *name* is a macro that expands into one of the two forms.

Macro replacement

`#define name replacement` defines an object-like macro that causes each subsequent instance of *name* to be replaced with *replacement*.

`#define name(parameters) replacement` defines a function-like macro that causes each subsequent instance of *name* followed by a (to be treated as an invocation of the macro. When defining the macro, the (must immediately follow *name*. When using the macro, the (may be separated from *name* by whitespace. Each argument specified in the macro call is expanded and used to fill in the *parameter* placeholder specified in *replacement*.

If a *parameter* placeholder in *replacement* is immediately preceeded by a `#`, then both are replaced by a string constant that contains the unexpanded spelling of the parameter. Leading and trailing white space of *parameter* is deleted. The backslash \ and double-quote " characters are escaped with a backslash character.

If a *parameter* placeholder is immediately preceeded or followed by a `##`, then the `##` operator is deleted and the preceeding and following tokens are concatenated. The unexpanded form of the *parameter* placeholder is used. The `##` operator can't be first or last in *replacement*.

Line control

`#line digits` resets the line number of the next source line to *digits*.

`#line digits string` resets the line number of the next source line to *digits*, and the name of the source file to *string*. *string* must be a string constant.

`#line name` behaves as the preceeding two forms, where *name* expands to one of the two forms.

Error

`#error text` produces *text* as a diagnostic message.

Pragma

Pragma directives consist of a name and an action. The **bold** portion of the name in the following specifications shows the minimum amount of the name that must be specified. The action can be + to turn the switch ON, - to turn the switch OFF, ! to invert the switch, and = to reset to the default state. No action turns the switch ON.

`#pragma trigraphs` controls trigraph processing. Default is OFF.

```
    #pragma t+
```

`#pragma warning` controls whether structure assignment, structure argument, and prototype argument type conversions are reported. Default is OFF.

```
    #pragma warn -
```

`#pragma extended` controls whether the extended keywords, including #asm, are recognized. Default is OFF.

```
    #pragma extended =
```

Null

A # on a line by itself is ignored.

Predefined macros

All predefined macros begin with two underscore characters '__', except for the LARGE_CASE macro (which is retained for compatiblity with previous releases).

`__LINE__`	is the number of the current source line (can be reset with `#line`).
`__FILE__`	is the name of the source file (can be reset with `#line`).
`__DATE__`	is the current date in the form `"Mmm dd yyyy"`
`__TIME__`	is the current time in the form `"hh:mm:ss"`
`__STDC__`	`"1"` = Standard-conforming implementation, `"0"` otherwise
`__DESMET__`	identifies this compiler
`LARGE_CASE` `__LARGE__`	indicates the Large-Case option, **b**, was selected.
`__SMALL__`	indicates the Large-Case option, **b**, was not selected.

EXTENSIONS

Asm

A #asm directive has been included to allow in-line assembly language code for time critical applications. All lines following a line starting with #asm are passed through to the assembler. A line consisting of the "#' character ends the in-line assembly code. Object-like macros (without arguments) as well as arguments and variables local to the function can be referenced by prepending a # to the name. Global variables are accessed by name with an '_' appended. Be sure to specify operand size (BYTE, WORD, ...). Extended keywords must be enabled (**px**).

```
#pragma ex+

#define COM1 0x3B4

int x;

_zip(int count, char *src, char *tar){
    int y;
#asm
    MOV CX,#count ;count
    MOV SI,#src   ;src
    MOV DI,#tar   ;dst

    mov WORD #y,#COM1
    mov WORD x_,#COM1
#
    }
```

Case range

As an alternative to coding

```
case 'a': case 'b': case 'c': case 'd': case 'e':
```

you can specify

```
case 'a' .. 'e':
```

Extended keywords (**px**) must be on.

RESTRICTIONS

Forward references

C88 is effectively a one pass compiler so forward references will not work. The following program:

```
main() {
   i=99;
   }
extern int i;
```

will produce a warning that 'i' is undefined and is assumed to be a local variable named 'i'. The global variable 'i' will not be changed.

Structure tags must be defined before being referenced. The only exception is pointers, so that legal structure declarations include structures of the form:

```
struct a {
    struct b *x;
    }

struct b {
    struct a *y;
    }
```

Externs

A declaration that includes the keyword 'extern' may not include initializers and does not allocate any memory. Thus a variable so declared must be declared somewhere else without the 'extern' keyword in order to reserve memory for the variable. For example, if a file contains the declaration `extern int blip`, then some other file must contain the declaration `int blip` to actually allocate storage. If this is not done, the binder will complain about a reference to the unresolved symbol *blip* . It is permissible to have both an 'extern' and non-'extern' declaration in a single file. For example,

```
extern int blip;
int blip;
```

is valid.

To create include files containing data declarations include the declaration:

```
    extern int blip;
```

add the declaration:

```
    int blip;
```

to one of the files to actually allocate the storage. If the variable needs initialization, initialize the value in the one file:

```
    int blip = 1985;
```

Large Case Option

With most programs, just compile with the -b switch on, and link using BBIND. Remember that **all** the object files must have been compiled or assembled with the -b switch.

Most of the difficulty in converting to Large Case is in the area of pointers. In Small Case, pointers and `int`'s are the same size — if you don't declare a function to return a pointer there is no harm done. The default `int` return of the function is the same size as a pointer.

In Large Case, however, pointers are four bytes and `int`s are two bytes long. You will get an error message if you try to assign an `int` or an `unsigned` to a pointer, or *vice versa.*

The macro names `__LARGE__` and `LARGE_CASE` will be true (defined) when the **-b** option has been specified on the command line. You can test for this condition using `#ifdef` or `#ifndef`

```
#if defined __LARGE__
     #define stdin  0L
     #define stdout 1L
     #define stderr 2L
#else
     #define stdin 0
     ...
#endif
```

The things to watch out for are:

- functions returning pointers must be declared before use.
- `fopen()` now returns a **pointer**, and must be declared before being called.

  ```
  FILE *fopen();        /* Assumes STDIO.H included */
  ```

  The pointer is used by `fclose()`, `fgetc()`, `fgets()`, `fprintf()`, `fputc()`, `fputs()`, `fread()`, `fscanf()`, `fseek()`, `fwrite()`, `getc()`, `getw()`, `putc()`, `putw()`, `ungetc()`;

  `open()` still returns, and the other I/O functions still use, an `int`. Note that this means that `fclose()` and `close()` are **not** interchangeable.

  This should make these functions more portable to other C environments.
- Large Case and Small Case object files cannot be linked together.
- A `long` can be assigned to a pointer, and *vice versa*.
- `malloc()` is slow as it calls DOS. This was done to leave as much space as possible free for calls to `exec()`.

The C88 Compiler supports both small and large case compilations. To compile a large case program, use the **b** switch, as

```
c88 blip -b
```

The hyphen is optional.

There are a few new Large Case Option error messages:

illegal indirection	something other than a pointer has been used as a pointer.
illegal index	a pointer cannot be used as an array index
illegal assignment	only a pointer, long, or constant can be assigned to a pointer. Note: this is a pass 2 error — the **-c** (checkout option) must be used to get the line number of the error.

Chapter 5

The ASM88 Assembler

Introduction 5.1

Invocation 5.1

Examples 5.2

Large Case ASM88 5.3

Introduction

ASM88 is the 8088/8086 assembler. It reads assembly language source files and produces linkable object files. The assembly language is described in appendix B.

Invocation

```
ASM88 <filename> [ options ]
```

<filename> is the name of the assembly language source file. If it does not include an extension — the extension '.A' is assumed.

Options: The case of the option is not significant. Each option should be separated from other options by blanks. Options may be preceded with the dash (-) character.

B The assembler creates Large Case (big) memory model object files. You need the Large Case Option (not included with the DeSmet C Development Package) to use this option.

L[<filename>] — The assembler will produce a listing from the assembly language input. This listing includes the hex-values generated by the assembler as well as line numbers and pagination. If no name is specified, then the name of the source file with the extension '.L' is used. If the specified file does not have an extension, '.L' will be used. Otherwise the listing is written to the specified file. To generate a listing on the printer, use '-LPRN:'.

M The assembler will produce an object file with the Intel formats rather than the standard .O format. The file TOOBJ.EXE from the DOS LINK package must be in the same directory as the GEN.EXE and ASM88.EXE files. You need the DOS LINK Option (not included with the DeSmet C Development Package) to use this option.

O<filename> — The assembler will produce an object file with the specified name. If the name lacks an extension, then the extension '.O' will be appended to the name. The default object file name is the name of the source file with the extension changed to '.O'.

T<drive> — The 'T' option specifies the drive where the assembler temporary files will be created. If a RAM Disk is available, redirecting temporary files to that drive will greatly speed development. The assembler normally creates its temporary files on the default drive/directory.

Pnn Specifies page length, in lines. The default is 66.

Wnn Specifies page width, in characters, for the list file. The value nn must be a number from 60 to 132. The default is 80.

Examples

```
asm88 blip
```

assembles the file named blip.a and produces a Small Case memory model object file named blip.o.

```
asm88 blip b
```

assembles the file named blip.a and produces a Large Case memory model object file named blip.o.

```
M:asm88 blip.asm -Ob:blip Lblip.lst
```

runs the assembler from drive M: on the file named blip.asm. The output is an object file named blip.o on drive B: and a listing file named blip.lst on the default drive.

```
asm88 blip.a TM -oa:blip.o -lb:blip.lst
```

assembles the file named blip.a. Temporary files are created on drive M:. The output of the assembler is placed on drive A: in the file blip.o. A listing file is generated and written to drive B: in the file blip.lst

Large Case ASM88

ASM88 also supports large and small case assembly. To assemble a large case program, use the **b** switch as

```
        asm88 blip -b
```

The hyphen is optional.

In addition to the standard CSEG (Code Segment) and DSEG (Data Segment) directives, there is a ESEG (Extra Segment) directive. CSEG and ESEG can be any size, while DSEG is restricted to 64K. CSEG is addressed with CS, DSEG with DS, and ESEG with either DS or ES. The Stack is a separate segment whose default size is 8K (changeable using the **-s** option of BBIND). The Stack is addressed by SS.

The long call and return, LCALL and LRET, are normally used instead of CALL and RET. You can mix the short and long forms of call/return in a program, but be sure that each form of return is matched to the corresponding form of call.

The DD directive creates a long (4 byte) pointer

```
        label DD  zip, zap
```

See Chapter 10 for the Large Case memory model layout.

All Assembly Language functions must preserve BP and DS.

There are two new prefix operators — `SEG` and `@`. `SEG` is similar to `OFFSET` except that it generates the segment of the variable rather than the offset. `@` is special — a long (4 byte) pointer is created (if needed) in DSEG and its offset is generated. `@` is normally used with `LES` to load a long (4 byte) pointer to a variable. For example:

```
                ESEG
        msg     DB      'Hello World!!',10,0

                DSEG
        msgptr  DD      msg

                CSEG
                PUBLIC  main_,puts_
        main_:  push    BP
                mov     BP,SP
        les     SI,msgptr       ; long ptr
        push    ES
        push    SI
        lcall   puts_
        mov     SP,BP

        mov     AX,seg msg      ; get segment
        mov     ES,AX
        mov     SI,offset msg   ; get offset
        push    ES
        push    SI
        lcall   puts_
        mov     SP,BP

        les     SI,@msg         ; seg:offset
        push    ES
        push    SI
        lcall   puts_
        mov     SP,BP

        push    @msg+2          ; seg
        push    @msg            ; offset
        lcall   puts_

        mov     SP,BP
        pop     BP
        lret
```

To facilitate writing assembler modules that can work with both Large and Small Case programs, the builtin symbol LARGE_CASE is recognized by ASM88. It has the value 1 if the -b flag is set, otherwise it is zero.

The control directives IF, ELSE, and ENDIF have been added to support conditional assembly. Any symbolic name — set by an EQU directive — can be used. For example:

```
                CSEG
                PUBLIC strlen_
        strlen_: push   BP
                xor    AX,AX
                mov    BP,SP
```

```
          IF     LARGE_CASE
          les    BX,[BP+6]        ; point to string
SL_LOOP:  cmp    BYTE ES:[BX],0   ; test for EOS
          ELSE
          mov    BX,[BP+4]        ; point to string
SL_LOOP:  cmp    BYTE [BX],0      ; test for EOS
          ENDIF

          jz     SL_RET
          inc    AX               ; length
          inc    BX
          jmp    SL_LOOP

SL_RET:   pop    BP

          IF     LARGE_CASE
          lret

          ELSE
          ret
          ENDIF
          END
```

When combining Large Case C88 and ASM88, keep the following in mind:

- Long calls (LCALL) and returns (LRET) are used.
- With the standard PUSH BP/MOV BP,SP prolog, parameters start at [BP+6]
- Pointers are returned in ES:SI
- Static and fundamental data are placed in DSEG, structures and arrays in ESEG

Chapter 6

The BIND Object File Linker

Introduction 6.1

Invocation 6.1

Examples 6.3

Small Case BIND
- Space Considerations 6.3
- Overlays 6.4

Large Case BIND 6.6

Libraries 6.7

Introduction

BIND is the program that links together object and library modules and forms an executable program. For very long command lines, see the **-f** option.

Invocation

```
BIND <filename> <filename> ... [options]
```

<filename> A sequence of filenames separated by blanks. The filenames should be the names of object (.O) or library (.S) files. If a filename does not have an extension, '.O' is assumed. BIND automatically looks for the supplied library CSTDIO.S so its name should not be included in the list of filenames.

Options: All options may be in upper or lower case. Options must be separated by blanks and preceded by a hyphen to differentiate them from <filename>s. Note that this is different from other commands where the hyphen is optional.

-A The assembler option keeps BIND from generating the C initialization code. Instead, execution begins at the beginning of the code rather than starting at the main_ public label. ARGC and ARGV are not calculated and the stack is not set up. Uninitialized variables are not filled with zero. Library functions such as creat() and open() cannot be used as they depend on the zero initialization. The 'A' and 'S' options are useful for a few cases but caution should be exercised in their use.

-C This option indicates that BIND should also generate a checkout (.CHK) file. This file is required when using the D88 debugger and the profiler.

-F<filename> identifies a file containing <filename>s and options to be used by BIND. This is used for very long lists of filenames and options.

-L<name> specifies the drive/directory containing the CSTDIO.S standard library. If this option is not specified, the CSTDIO.S file must be on the default drive. With MS-DOS 2.0 and later versions of the operating system, the PATH system parameter is used to locate the library.

-Mn Indicates that the object files following this control should be collected in the memory-based overlay indicated by the value n (1 to 39). See the description on overlays below for details on the overlay mechanism.

-O<filename> changes the name of the output file to <filename>.EXE. If this option is not specified, the name of the first object file in the list with the .EXE extension will be used.

-P[<filename>] Generates a sorted list of publics and offsets. C procedures and data declared outside of procedures are automatically public (or extern) unless explicitly declared static. Publics with names starting with an underline '_' are not listed unless the -_ option is also specified. The optional name is the destination for the publics list. If omitted, the publics and offsets are listed on the console. The size of overlays, if any, will also be displayed.

-Shhhh Specifies the stack size. **hhhh** is in hex. Normally, BIND will set the stack size as large as possible for the Small Case memory model, and to 8K bytes for the Large Case memory model.

-Vn This option is used to create disk-based overlays. All object files following this option, until the end of the list or another overlay option, are collected into the overlay indicated by the value n (1 to 39). See the overlay section below for details.

-_ (underscore) — BIND normally suppresses names that start with an underscore (usually internal names) from the *publics* list and the .CHK file. The underscore option makes these names available. This option is required when you need to see all the modules bound to your program.

Examples

```
bind blip
```

binds the file *blip.o* with CSTDIO.S and produces the executable file *blip.exe*.

```
bind proga progb progc lib.s -p
```

binds the files *proga.o*, *progb.o*, and *progc.o* with the user library *lib.s* and the standard I/O library, CSTDIO.S, into the application file *proga.exe*. The map is printed on the screen.

```
bind proga progb -V1 progc -V2 progd -Pmap -_ -Omyprog
```

binds the files *proga.o*, *progb.o* with CSTDIO.S and creates the executable file *myprog.exe* and the overlay file *myprog.ov* which contains two overlays consisting of the object files *progc.o* and *prod.o*. The publics map is sent to the file named *map* and will also list the internal names that begin with the underline ('_') character.

Small Case Bind

Space Considerations

A program is restricted to a maximum of 64KB of code and 64KB of data plus the stack. BIND calculates the size of code and data and will report the size of each segment (in hex) when the -P option is specified. BIND cannot calculate the actual stack requirements. If the 'stack' and 'locals' size reported by BIND seems small, the actual stack requirements should be calculated by hand to make sure there is enough space. The actual requirements are the worst case of four bytes per call plus the size of locals (including parameters) for all active procedures plus about 500 bytes for the Operating System calls. In practice, 2KB plus the size of the local arrays simultaneously active should be sufficient.

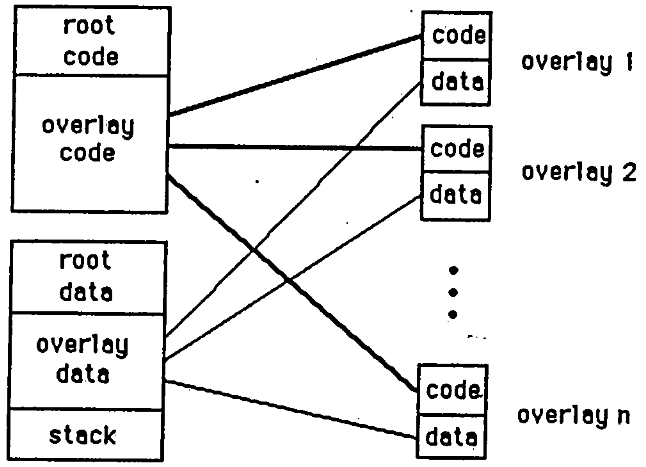
If BIND reports that the code limit is exceeded, look in the publics map for the scanf() and printf() routines. These are relatively large routines (around 2KB each) and also link in the floating-point routines. Eliminating the use of these routines can result in a large savings. If scanf() and/or printf() are necessary but no floating-point values will be used, try using the CSTDIO7.S instead of the standard

CSTDIO.S library (Rename the CSTDIO.S library to something else and rename the CSTDIO7.S library to CSTDIO.S). This will assume the availability of the 8087 math chip and will not bring in the software floating-point routines.

Another way to save some space is to use the CREAT2.C file from the optional HACKERS disk (not distributed with the compiler) which contains a version of the I/O routines open(), close(), etc. that only work with MS-DOS 2.0 and later versions of the operating system. This saves around 3KB but will not allow the program to be run under MS-DOS 1.xx.

Overlays

Another way to solve the space problem is the use of overlays. The overlay system provided by this package is very simple. An application is divided into a root portion that is always resident and two or more overlays. Only one overlay is resident (executable) at any given time. The following diagram outlines the relationship between the root and the overlays:



There are two types of overlays, disk-based overlays and memory-based overlays. The difference between the two types is the location of the overlays. Disk-based overlays, created with the -V option, are stored in a separate file. Memory-based overlays, created with the -M option, are loaded into memory along with the root code. Memory-based overlays should only be used when there is sufficient memory for the root and all of the overlays. The advantage of memory-based overlays over disk-based overlays is in the amount of time needed to make an overlay resident, memory-based overlays being much faster to load.

The application program is responsible for initializing the overlay subsystem and ensuring that the correct overlay is resident before calling any of the functions in the overlay.

For disk-based overlays, the routine `overlay_init()` must be called from the root with the name of the overlay file to initialize the overlay system. Overlays are loaded by calling the routine `overlay(n)` where n is the number of the overlay to be made resident.

For memory-based overlays instead of disk-based overlays, do not call the `overlay_init()` routine and call the routine `moverlay()` in place of the routine `overlay()`.

In the following example the root is composed of the file X.C. The first overlay is the file Y.C and the second overlay is in the file Z.C.

File X.C:
```
     main() {
          overlay_init("X.OV"); /* initialize */
          puts("this is the root program\n");
          overlay(1); /* make 1st overlay resident */
          zip();      /* call into 1st overlay */
          overlay(2); /* make the second resident */
          zap();      /* call into second overlay */
          puts("bye\n");
          }
```

File Y.C:
```
     zip() {
          puts("  this is ZIP  ");
          }
```

File Z.C:
```
     zap() {
          puts("  this is ZAP  ");
          }
```

The files are compiled in the usual fashion:

```
    c88 x
    c88 y
    c88 z
```

Ordinarily, the files would be linked together using the command:

```
    bind x y z
```

Instead, to create the two overlays, the command:

```
    bind x -V1 y -V2 z
```

is used. The -V option is followed by the overlay number. This number starts at 1 and runs in ascending order up to 39. All files following the -V or the -M option are included in the overlay. All library modules (from .S files) are included in the root. The result from the execution of the BIND program with the -V option is the executable root (.EXE) file and the overlay (.OV) file which contains the overlays. The result with the -M option is an .EXE file containing both the root and the overlays.

D88 knows about the overlays and will not display public symbols that are not resident. The profiler does not know about overlays and should not be used.

The -P option of BIND will also display the size of each overlay as well as the overlay for each symbol.

Large Case BIND

The Large Case Binder's name is BBIND. In most respects, BBIND is identical to BIND. The differences are:

- BBIND only works with Large Case object files and libraries.
- BBIND uses BCSTDIO.S instead of CSTDIO.S. Rename BCSTDIO7.S to BCSTDIO.S if you use an 8087.
- The default stack size is 2000H (8096) bytes. This should be more than enough unless you have a huge amount of local data. The stack requirements are six bytes plus the local data space required for each active function call. You can change the stack size with the **-s** option.
- Overlays are **not** supported.
- The **-p** (Publics) map displays four byte addresses.

Libraries

Libraries are just concatenated .O files. The .S extension tells BIND to only include modules that are referenced. If all of the routines in a library are required, rename the .S file to a .O file to force all of the modules in the library to be included.

BIND includes the entire .O module from a library if any of its public names have been selected by other object modules processed by BIND. Thus, if a .O file contains several functions, all of them will be bound into a program if any of them are called.

BIND searches a library once. Thus if you have two modules, *A* and *B*, and *A* calls *B* , the *B* must follow *A* in the library. LIB88 attempts to order the library so that these inter-library references are ordered so that BIND will find them. One way around any circular dependencies (e.g., *B* also calls *A*) is to include the library twice on the command line.

Chapter 7

The LIB88 Object File Librarian

Introduction 7.1

Invocation 7.1

Examples 7.2

Libraries 7.2

Introduction

LIB88 is the program that combines object modules into library modules. Libraries are simply collections of object files in a single file from which the BINDer can select the necessary modules. By using a library, only those modules required by an application will be bound into the executable (.EXE) file.

You can't mix Small Case memory model and Large Case memory model object files in the same library.

Invocation

LIB88 <filename> <filename>... [option]

<filename> names of object files or other libraries. If no extension is given on the filename, '.O' is assumed.

Options The case of the option is not significant. Each option should be separated from other options by blanks. Options **must** be preceded by the minus sign ('-') character to distinguish them from <filename>s.

-F<filename> the pathname of a file containing filenames and options to be used by LIB88. This is used to get around the 128 character command line limit.

-N forces all input modules to be included in the output even if publics clash. Normally when there are duplicate public symbols, the module with the first occurrence of the symbol is kept; all others are ignored.

-O<filename> supplies the name of the target library. No extension should be included as LIB88 will add the extension '.S' which is required for a library. If omitted, the first filename forms the basis for the library name.

Caution: if a library (.S) file is first on the LIB88 invocation, the -O option must be used or no library will be created. The <filename> cannot be the same as the .S name.

-P[<filename>] A list of code publics is produced. The list goes to the named file if present, otherwise to the console. Data publics are not included in order to make the list shorter. A minus sign is in column 1 at the start of each module.

-_ (underscore) Publics that start with underscore are normally omitted from the publics list. The underscore option will include them.

Examples

```
LIB88 xx yy zz -Oxlib
```

combines the object files xx.o, yy.o, and zz.o into a library named xlib.s

```
LIB88 xx -Fblip
```

where blip contains

```
    yy zz
    -Oxlib
```

behaves exactly the same as the first example.

```
LIB88 xx xlib.s -Oylib
```

replaces the object file xx.o in the xlib.s library and places the result in a new library named ylib.s.

Libraries

Libraries are simply collections of object modules that are included into a program by BIND as necessary. A library is only searched once by BIND so if a library member A calls library member B, module B must follow module A in the library. The librarian will attempt to sort modules so the caller comes first in the target library. If modules call each other, LIB88 will print the warning

```
    circular  dependencies
```

The -N (for need) option is used to force object files in a particular order (ignoring circular dependencies). It ignores the LIB88 sort logic and concatenates all the <filename>s into a library.

LIB88 installs the first occurrance of a PUBLIC name into the target library. Thus if two modules have PUBLICs in common, then the module encountered first will be installed in the library. Thus to replace the CSTDIO.S version of `qsort()` with your own, you would do the following

```
c88 qsort
ren cstdio.s cstdio.o
lib88 qsort cstdio -ocstdio
del cstdio.o
```

CSTDIO.S was renamed to CSTDIO.O to avoid any conflict of reading and writing to CSTDIO.S during the update.

LIB88 cannot replace object modules in libraries with circular dependencies. To update libraries that have circular dependencies, use both the -F option to name the file of module names, and the -N option to suppress LIB88 sorting.

Libraries are just concatenated .O files. The .S extension tells BIND to only include modules that are referenced. If all of the routines in a library are required, rename the .S file to a .O file to force all of the modules in the library to be included.

Chapter 8

The D88 C Language Debugger

Introduction	8.1
D88 Usage	8.1
Command Input	8.3
Expressions	8.3
Commands	8.5
Again	8.5
Breakpoint	8.5
Collection	8.6
Display	8.7
Expression	8.8
Flip	8.8
Go	8.9
List	8.10
Macro	8.11
Options	8.12
Proc-step	8.13
Quit	8.14
Register	8.14
Step	8.14
Unassemble	8.15
Variables	8.15
Where	8.16

Introduction

D88 is the C source language debugger for C88. Its features include:

Full screen display.

C source can be displayed while executing.

All local and global variables can be displayed.

C expressions can be evaluated.

Special support for debugging interactive programs on the PC.

Breakpoints by address or line number.

D88 only works with programs produced by the C88 compiler because it needs special symbol, type and line number information. It is not as good as DEBUG when dealing with assembler programs. Like all debuggers, D88 needs lots of memory -- about 45K extra for small programs and 64K for large ones. For systems that are not IBM PC compatible, D88 will have to be configured before it can be used. See the the instructions in the CONFIG.C file on Disk #2.

CAUTION: do not change floppy disks while D88 is executing. Changing any disk while a program is running may clobber the new disk.

D88 Usage

D88 needs symbol information that is not normally created. Before using D88, a program should be compiled and bound with the 'C' option in order to create the symbol information. You can bind in modules that were not compiled with the 'C' option, but their symbol and line number information will not be available. Assuming that the C88 compiler, binder, library (CSTDIO.S) and D88 are on drive A: and that the D88 sample program CB.C is on drive B: the following commands will compile CB.C and create the symbol file.

```
A>C88 B:CB -C
  C88 Compiler    V2.5    (c) Mark DeSmet, 1982,83,84,85
  end of C88        04B7 code   00D7 data       21% utilization

  A>BIND B:CB -C
  Binder for C88 and ASM88 V1.9(c) Mark DeSmet,1982,83,84,85
  end of BIND         19% utilization
```

The 'C' options will create the checkout file CB.CHK in addition to the usual executable CB.EXE file. The CB.CHK file contains pathnames so the user should invoke D88 with the same default drive (and current directory with MS-DOS V2.xx, ...) that was in effect during compilation so that D88 can find the C source.

The CB.C program is executed by

```
B:CB filename
```

For example, to run CB on itself:

```
 A>B:CB B:CB
231 lines
A>
```

No errors were detected. To debug or trace a program, prefix the normal execution line with D88. Using the above example:

```
D88 B:CB B:CB
```

D88 will clear the screen, print the banner and issue the following prompt.

```
Again Breakpoint Collection Display Expression Flip Go --space--
-----
procedure MAIN  file B:CB.C  line 18
-----
D88 Debugger V1.4      (c) Mark DeSmet 1984,85
```

D88 command input is similar to SEE command input. The top line contains a partial list of available commands. To see the rest, hit the space bar and the top line will change to the next prompt line. The prompt lines are:

```
Again Breakpoint Collection Display Expression Flip Go --space--
List  Macro  Options  Proc-step  Quit  Register  Step  --space--
Unassemble  Variables  Where  --space--
```

Error messages are displayed on line two. The fourth line gives the name of the current procedure, the name of the current source file and the current line number. This line is always displayed. The remainder of the screen scrolls in the usual fashion.

Command Input

As with the SEE editor, commands are entered by typing their first letter. For example, typing 'R' will display the registers. Commands may be entered in upper or lower case and the command need not be displayed on the prompt line to be executed. The Again, Display, List and Unassemble commands can be preceded by a decimal repetition count. The count specifies how many lines should be processed.

If a command has options, a prompt is issued to ask for them. For example: type 'L' for List and the following prompt will be issued on the top two lines.

```
 enter list line number or search string
exchange: 18
```

The cursor will be at the first letter of '18'. Typing return or ESC means the number is correct. To change it, the number may be overtyped or edited with the following keys.

->	The right arrow moves the cursor to the right.
<-	The left arrow moves the cursor to the left.
Ins	The Ins key toggles between Exchange mode and Insert mode. The prompt changes between 'exchange:' and 'insert:'.
Del	Deletes the character under the cursor.
backspace	Deletes the character to the left of the cursor.

When any editing is complete, press Return or ESC. During input, type control-C to abort the command and return to the main prompt.

Expressions

Several commands will accept expressions. Expressions follow the usual C rules and are composed of variables and constants combined by operators.

Variables can be referred to by name; case is ignored by D88. Only extern or static variables, local variables in the current procedure and parameters of the current procedure can be referenced. There is no way to reference locals of another procedure. Statics are not scoped by file -- the first entry in the symbol table is used. Statics that are defined within a procedure have their name prefixed by the procedure name and '_', e.g. static int i; in main is called MAIN_I. The Variables command will list the names of variables and the Expression command will display their values.

Examples: `argc    i    nextin    main_i`

Registers may be referred to by name. Example: `ax`

Constants may be of type `int, long` or `float`. Hex constants must start with '0' but must omit the 'x'. Octal constants are **not** permitted. Strings and character constants are as usual. Examples: `2   23.6   1e6   01abc   'A'   "hello world!"`

Member references may follow the '.' or '->' operator.

Examples: `stru.mem   sptr->mem`

Most of the usual C operators are supported. They are listed below in order of precedence.

assignment	=
addition, subtraction	+ -
multiplication, division, modulus	* / %
contents of	*
address of	&
prefix minus	-
array	[ ]
parenthesis	( )
function call	name()

Examples of expressions:

```
    2+2   *argv[1]   stru->mem   &vara   ax=44  "hello"[2]  printf("%d",2+2)
```

The last example shows that functions in the program under debug can be executed by the Expression command. An expression followed by (arguments) will be called but referring to a function name not followed by the '(' yields the offset of the function.

Commands

To learn D88 try out all of the commands on the CB program. One caution: when a function is first entered, locals and parameters cannot be accessed until you use the Step command to move down to the first executable instruction.

[n] Again — only has meaning after a Display, List or Unassemble command. It displays the next n lines of bytes, source lines or disassembled instructions respectively. If the count is omitted, 10 lines of source or 3 lines of bytes or disassembled instructions are displayed.

Prompts: none.

Output: Depends upon prior command.

Breakpoint — sets a 'sticky' breakpoint. A breakpoint is a place where execution will stop after a Go command. A 'sticky' breakpoint is one that remains in effect until changed or the Quit-Init command is entered.

Prompts:
```
enter number of sticky breakpoint, 1 2 or 3.
```

There can be up to three sticky breakpoints, numbered 1,2 and 3. Enter the number of the breakpoint you wish to change.

```
Address-break Line-number-break Procedure  Forever
```

Enter **A** or **P** if you want to break at an address or procedure. The next prompt will be:

```
    enter procedure name or address
```

Enter an expression that indicates where you wish to stop, e. g. `puts` or `0a1c`.

Enter **L** if you want to stop at a specific line number. The next prompt will be :

```
input line number
```

The file is the current file unless changed by the Options-Listfile command. There is no default line number. Only line numbers for lines containing executable instructions can be referenced. You cannot break at a declaration or comment.

Enter **F** for Forever to remove a sticky break or Go to completion.

[n] Collection — displays the elements of an array or structure. The optional repetition count is the number of array elements that will be displayed. If a member is specified, that and all subsequent members of the structure will be displayed. The display format is the same as that described under the description of the Expression command.

Prompts: `input an array name or structure.member.`

Output: Assuming the following program,

```
char a[5]={1, 2, 3, 4, 5},
     b[3][5]={1,2,3,4,5,6,7,8,9,10,11,12,13,15},
    *c=&a;
struct {int i,j,k;} str={11,22,33},
      *st=&str;
main() {;}
```

The following collections can be displayed.

```
input an array name or structure.member
exchange: a
array at 0004
[0]=  1    [1]=  2    [2]=  3    [3]=  4    [4]=  5

 input an array name or structure.member
exchange: b
array at 0009
[0]=array at 0009   [1]=array at 000E   [2]=array at 0013

input an array name or structure.member
exchange: b[1]
array at 000E
[0]=  6   [1]=  7   [2]=  8   [3]=  9   [4]= 10
```

```
input an array name or structure.member
exchange: c
0004->
[0]=  1    [1]=  2    [2]=  3    [3]=  4    [4]=  5    [5]=  1
[6]=  2    [7]=  3    [8]=  4    [9]=  5

input an array name or structure.member
exchange: str
structure at 001A

input an array name or structure.member
exchange: str.i
.I=     11 000B   .J=     22 0016   .K=     33 0021

input an array name or structure.member
exchange: st->i
.I=     11 000B   .J=     22 0016   .K=     33 0021
```

The examples demonstrate the following rules:

1. If an array name is entered, the address of the array is printed followed by the first 10 (or repeat) elements.

2. A pointer is handled the same way except that the number of elements is not known. Notice that arrays used as parameters are passed as pointers so the number of elements is not known.

3. If the name of a structure element is entered, that and all subsequent members are displayed. Either the '.' or '->' operator may be used as appropriate.

4. If any other type of expression is entered, the value is displayed.

See the Expression command for the rules for element display.

[n] Display — displays memory in hex and ASCII. In contrast to Expression, types are ignored. The optional repetition is the number of lines to display. The default number of lines displayed is three.

Prompts: `input [segment:] offset`

Normally a pointer name is input to see what it points to in hex. Notice that if a variable name is input, the variable value is used (e.g. if i is 3 then a Display of i is the same as a display of 3). Use the address (&) operator to see how a variable looks in hex — &i would display i in hex. The data segment is always assumed. Use an override to display other segments e.g. cs:0.

Output:

```
75B8:07BE 2F 2A 09 43 42 43 48 45 43 4B 2E 43 20 20 2D 2D
75B8:07CE 09 44 75 6D 62 20 43 75 72 6C 65 79 20 42 72 61
75B8:07DE 63 65 20 43 68 65 63 6B 65 72 20 66 6F 72 20 43
```

Expression — evaluates and displays the results of an expression. A procedure can be executed by including its name and parameters in an expression -- be careful of side effects. Only a subset of the normal C operators is supported but otherwise expression rules for precedence, pointer arithmetic and type conversion apply. The assignment operator can be used to set a variable or register. Static variables within functions have their name preceded by the procedure name and an underscore.

```
zip() { static int i; }
```

'i' would be referred to as 'zip_i' in the debugger. Examples:

2+2 argc argv[1] nextin bp+4 puts("hello!") puts ptr->off i=44

Prompts: `input an expression`

Output: Chars are displayed in unsigned and ASCII if possible, e.g.

'C' 67.

Unsigned are displayed as unsigned and hex.

A pointer is displayed in hex. In addition, the string '->' prints and the element pointed to are displayed. In the case of a pointer to a character, up to 21 characters are displayed on the assumption that the pointer is to a string.

Ints are displayed as decimal and hex.

Float and double are displayed as %9.2E.

Longs are displayed in decimal.

Arrays are displayed as 'array at' hex address.

Functions and structures are similar to arrays.

Flip Debugging graphic or full screen applications can be a real problem as both debugger and application need to use the screen and the two displays interfere with each other. The Flip command is part of the mechanism designed to deal with this problem.

The Flip command will flip the screen. It only works on PC compatibles as it is hardware dependent (see notes in CONFIG.C and FLIP.A on configuring this capability). The idea is that the user should have two screen displays — one which is produced by the program under debug and the other which is used for the D88 display. The application screen is automatically restored before the Go command resumes execution. The Flip sub-option of the Step and Proc-step command must be used to restore the application screen before executing any command that affects the screen display. When the screen image is preserved in this way, the Flip command can be used to display the application screen. Pressing any key will return to the D88 screen.

Prompts: none.

Output: The application screen will replace the D88 screen. Hit any key to return to D88.

[n] Go causes the program being debugged to execute. The user is prompted to enter one breakpoint. The description of the Breakpoint command describes how this breakpoint may be entered. The breakpoint may be at the current address; if you enter an address breakpoint of IP, the program will execute until it returns to the starting point. This can be used to execute one iteration within a loop.

After a Go command, 9 lines of the source are displayed. A '->' points to the current line. The Option command can turn this feature off.

The optional repetition specifies how many breakpoints should be hit before execution ceases. A count of 10 Go's to IP would execute a loop 10 times.

If the Option command sets the 'Flip on Go' option off, the output of the debugged program and D88 output will be intermixed. The default is to display the debugged programs output before execution commences.

Once started, a program will execute until a Breakpoint break is hit, the Go breakpoint is hit, EXIT is called or control break is hit. In any event the Go breakpoint will be removed. Under DOS 1.x (and CP/M-86), EXIT and control break will cause D88 to terminate. Under MS-DOS 2.xx, ..., 'normal end' prints and D88 continues. The Forever option should be used if you wish the program to run to completion or to a sticky break set by the Breakpoint command.

Prompts: `Address-break  Line-number-break  Procedure  Forever`

See the description of breakpoint entry under the Breakpoint command description.

Output: The program will execute.

[n] List lists any ASCII file. It is normally used to list the source of the program being debugged. If the count is omitted, 10 lines will be listed. After a List command, the Again command can be used to list more lines without entering the line number.

The current file is the one listed unless the Options-Listfile command is entered.

The prompt asks for the line number or a string. If something other than a number is input, then the List command only lists lines that contain the characters. Searching always starts from top of the file. The search string option can be used to find a procedure definition or variable references.

Prompts: `enter list line number or search string`

The default is the current line number or the last line listed if the List command was just executed. Enter return to list source from the current line or a decimal line number or a search string.

Output:

```
enter list line number or search string
exchange: 18
      18  main(argc,argv)
      19      int argc;
      20      char *argv[]; {
      21      int  ch;
      22      char col;
      23
      24      if (argc < 2) error("no file name","");
      25      read_file(argv[1]);
      26
      27      while (1) {

enter list line number or search string
exchange: read_file
      25      read_file(argv[1]);
      47  read_file(fil)
```

Macro

remembers commands or sequences of commands. Four Macros can be defined — F1, F2, F3 and F10. All keyboard input is collected into a Macro until another Macro command is entered. Once defined, a Macro is executed by simply hitting the appropriate function key. A Macro can be up to 80 keystrokes long.

Prompts:

```
enter name of macro. F1 F2 F3 F10
Hit the appropriate function key.
```

Output:

```
enter another Macro command to end definition
```

Printed after the above prompt is answered. All input will be accumulated into a Macro until another Macro command is entered.

```
Macro is defined
```

Printed if the Macro command is invoked to end a Macro definition.

F10 is a 'permanent' Macro. If defined, it is run every time the screen is re-written and its output is placed after the top 3 lines. This permits variables to be permanently displayed.

For example:

`M`	(hit M for macro command)
`F10`	(hit function key 10 as name of macro)
`E`	(hit E for expression command)
`"i=",i,"j=",j,"k=",k`	(enter expression — note the comma means a list of values)
`M`	(end macro definition)

A line like `i= 44 j= 2 k= 11` will be displayed near the top of the screen until F10 is redefined. The values are thus continually updated.

Options

There are currently three options: flip screen on go, list after go, and list file name.

The Flip-on-go option allows D88 output to be intermixed with user output. The default is to flip the screen before a go executes. The disadvantage of not flipping is that the output of the application will be intermixed with D88 output. The disadvantage of the default is the flashing that occurs if the Flip is not needed.

The Go-list option can disable the listing of source after a go command. Every Go, Proc-step and Step command sets the current listfile name to the file containing the current statement. This name is used by the List, Breakpoint and Go commands. Use the List-name option to change the name.

Prompts: `Flip-on-go  Go-list  List-name`

If **F** is typed,

```
     flip screen on Go (y or n) ?
```

Enter 'Y' or 'y' to set the option on, 'N' or 'n' to turn it off.

If **G** is typed,

```
     list after a Go (y or n) ?
```

Enter 'Y' or 'y' to set the option on, 'N' or 'n' to turn it off.

If **L** is typed,

```
input list file name
```

Type the desired filename.

Output: none.

Proc-step

The Proc-step command prints the current source line and allows the user to execute it. Proc-step differs from Step in only stopping on lines in the current procedure. Proc-step also stops after a return so you can Proc-step back to the calling procedure. Step will stop on any line.

A Flip option allows the user screen to replace the D88 screen during stepping. If this option is not invoked before statements that affect the screen, then program output will be intermixed with D88 output. When the screen is flipped, there is no prompt but the user must still hit space to execute the next statement. Typing 'F' for Flip will restore the D88 prompt.

Only executable lines will show up while stepping; declarations and comments are not listed.

The `procedure MAIN  file B:CB.C  line 18` line is updated during stepping.

Prompts: `Flip  Proc-step Step  space to Proc-step. default=quit.`

`F` will flip the screen. `S` will change from `Proc-step` to `Step` and step the current line. `P` will change back to `Proc-step` and step the line. Space will `Proc-step` or `Step`, whichever is current. If the screen is not flipped, the next line will print. Typing anything else will terminate stepping.

Output: When the screen is not flipped, the current line prints as a prompt.

Quit

The Quit command terminates a debugging session and either exits to the operating system or starts a new session. On exit, the user screen is restored.

The Initialize option allows debugging to begin again. **Caution:** files are not closed. You may run out of files or not be able to re-open files.

Prompt:

```
Quit:  Exit  Initialize
```

E restores the user screen and returns to DOS. If the program has been `EXIT`ed or interrupted with control break, you can only `Exit`. I displays the following prompt:

```
    input command line
```

Enter the part of the command tail that would follow D88, if D88 were being executed; e.g.

```
    CB CB.C
```

Press the return key if you change your mind and do not want to start over.

Output: D88 quits or starts over with the indicated command tail.

Register

The Register command displays all the registers. Use the Expression command to set a register to a value.

Prompts: none.

Output:

```
AX=7500 BX=FFEB CX=0000 DX=0000 SI=FFFF DI=07BE BP=FF8E SP=FF90
DS=757E SS=757E ES=757E CS=729E IP=0003 FL=F206
```

Step

The Step command prints the current source line and allows the user to execute it. Step differs from Proc-step in stopping at every line — not just lines within the current procedure. If you step a line that contains a call to another procedure, you will step through the called procedure. See the description of the Proc-step command for details on this command as Step and Proc-step are otherwise identical.

[n] Unassemble — The Unassemble command disassembles some instructions. The repetition count says how many instructions should be disassembled. The default is 10. The Again command can be used after an Unassemble command to print more instructions without re-entering address. Disassembled output follows normal assembler rules except that relative jumps print their target as absolute numbers (A=hhhh).

If the repetition count is '/', the Unassemble command will disassemble one line and prompt with a'?'. Pressing the space key causes the instruction to execute. This continues until the user presses a key other than the space key

Prompts: `input [segment:]offset`

The default address is the current one. If an expression is entered, it is assumed to refer to an instruction in CS:. An explicit segment can be entered, e.g. `0123:0da`.

Output:

```
729E:0003  55        PUSH  BP
729E:0004  8B EC     MOV   BP,SP
729E:0006  83 EC 04  SUB   SP,0004
```

Variables The Variables command will list the program variables, optionally with values. Pressing return to the prompt will produce a four across list of all variable names. The locals accessible to the current procedure are listed first, followed by the publics. Both are sorted. If a name or name pattern is entered, the variables are listed with their value. The values are formatted according to the rules for the Expression command. An asterisk ('*') at the end of a name means match any name that starts with the preceding letters. An asterisk by itself will list all variables with values.

Caution: before the first instruction of a procedure is executed, the stack frame is not established and parameters will not be printed correctly.

Prompt:

```
input variable name or pattern (a* means start with a)
```

Output:

```
input variable name or pattern (a* means start with a)
exchange:

ARGC        ARGV
ATOI        ATOL        A           B
CI          CO          CSTS        C
EXIT        GETCHAR     III         II
INDEX       I           JJJJJ       JJJJ
JJJ         JJ          J           MAIN
PUTCHAR     PUTS        RINDEX      STRCAT
STRCMP      STRCPY      STRLEN      STRNCAT
STRNCMP     STRNCPY

input variable name or pattern (a* means start with a)
exchange: i*

III =       5 0005
II =       3 0003
INDEX = function at 031E
I =       2 0002
```

Where

The Where command list the current procedures. The name, file and line number of every procedure currently executing will print.

Prompts: none.

Output:

```
procedure READ_FILE      file CB.C  line 56
procedure MAIN           file CB.C  line 25
```

Chapter 9

Utility Programs

CLIST: a listing & xref utility	9.1
DUMP: a hex and ascii display utility	9.2
FASTSCR: a screen output enhancer	9.3
FREE: a free space display	9.3
GREP: a file search utility	9.3
LS: a directory listing utility	9.4
MERGE: a C source and assembly language merge utility	9.5
MORE: a file screen listing utility	9.5
PCmake: a program maintainance utility	9.6
PROFILE: a performance monitor utility	9.9
RM: a file removal utility	9.11
SENSE87: an 8087/80287 sensing library	9.12
TOOLBOX.S: a library of useful tools	9.14

CLIST: a listing & xref utility

The clist utility reads C source programs and produces a listing, or a file, which contains a paginated, line numbered listing of the C source lines and a symbol cross-reference map.

To invoke the clist program, enter

```
clist <filename> ...   [ option ]
```

<filename>... — a list of the C source files to be listed, in the order that they are to be listed. If no extension is given, '.C' is assumed. Clist does not automatically read the "include" files so they should be listed first. This is to prevent the include file from being listed by every source file that "includes" it. Note: if you specify more than one file, the symbols will be combined into one cross-reference map.

Options: The case of the option characters is not significant. Each option must be preceded by the minus-sign, '-', character to distinguish it from a filename. The options are:

-F<filename> — identifies the file containing the <filename>s to be listed.

-L<size>/-P<size> — sets the page length used by the clist program for generating pagination. The default is 66 lines.

-N eliminates the cross reference listing

-O<filename> — supplies the name of the output file for the listing. Without this control, the first name in the list of filenames is used with the extension, '.L'. If no extension is given on the filename, '.L' will be used automatically. If you wish to list on the printer use -OPRN:

-T<size> sets the width for tab characters (the maximum number of spaces that a tab occupies). The default expansion size is 4.

-W<size> sets the width of the listing. Lines wider than the width are wrapped to the the next line. The default width is 80.

For example

```
    clist blip.c
```

will generate a file named `blip.l` with the following contents

```
BLIP.C   dd/mm/yy  hh:mm:ss                  Page 1
1   main() {
2        int i;
3
4        printf("Table of Characters\n");
5        for (i = 0; i < 256; i++) {
6             printf("Character %d prints as %c\n",i,i);
7             }
8        }

----XREF----

i        2   5   5   5   6   6
main     1#
printf   4   6
```

The symbol, '#', following the 'main' symbol in the cross-reference listing indicates that the symbol was declared on that line. Clist only supplies the declaration line information for procedures or data declarations which begin in the first column.

Dump: a hex and ascii display utility

The dump utility program is used to display the contents of a file in hex. It is available in both source (.C) and executable (.EXE) form.

To invoke the program, enter

```
          dump <filename>
```

The dump program displays 16-bytes per line with each line showing the offset to the first byte in the line, the 16 hex values, and the character equivalent enclosed between asterisks (*). For example:

```
0000 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 *ABCDEFGHIJKLMNOP*
0010 51 52 53 54 55 56 57 58 59 5A 00 00 00 00 00 00 *QRSTUVWXYZ......*
```

FASTSCR: a screen output enhancer

This is a simple program to make character output to the screen run faster than the standard IBM-PC BIOS routines. It works by intercepting the level 16 interrupts (int 10H) and routing the character output(write tty) requests to code which acts directly on the screen buffer. Requests which are not supported are passed on to the normal BIOS routine.

Works for all programs which use the BIOS calls or DOS calls.

To install the program, simply run

```
   fastscr
```

FREE: a free space display

free is a very simple program to show the amount of free space left on a drive without having to watch a directory listing. The syntax for free is:

```
    free <device>
```

where <device> is A: B: ...

GREP: a file search utility.

grep is a program used to search files for lines containing a certain pattern. Whenever the pattern is found, the line is listed on the screen. The syntax for grep is:

```
     grep [ -y ] <search pattern> file ...
```

`-y`	indicates that case is to be ignored during the search.
<search pattern>	is any sequence of characters. It may contain the wildcard characters * and ?. NOTE: if the pattern contains blanks or wildcard characters, then the search pattern must be enclosed within double quotes (").
file	is any filename and may contain wildcard characters.

When a match is found, grep will display the line along with the line number and filename.

LS: a directory listing utility

ls is a directory listing program loosely based on the UNIX utility. It features multiple-column listing, sort by name or modification date and reverse order sorting. The syntax for running ls is:

```
    ls [ -ltr1? ] [ pathname ... ]
```

`-l` invokes the long format which is a single column listing with the following format:

```
        <name>  <attrib>  <size>  <time>  <date>
```

`<attrib>` has three character fields. The first field displays the type of file. The second field indicates read permission. The third field indicates write permission.

TYPE	READ	WRITE	
-			= normal files
s			= system files
d			= directories
h			= hidden files
	r		= readable
		-	= not writeable
		w	= writeable

`-t` changes the sort order to list files by their modification dates rather than the default alphabetical order. Times are listed with the most recently modified files first.

`-r` reverses the order of the sort, alphabetic or by time.

`-1` produces a single column of output.

`-?` displays the input syntax for ls.

pathname can be a drive name (C:), a directory name, or a filename which may contain the wildcard search characters * and ?. If no pathname is given, the current directory is assumed.

At the end of the listing, the number of files listed and the total size of the listed files is displayed.

MERGE: a C source and assembly language merged listing utility.

merge is a program used to merge a C source file with its corresponding assembly language file generated by C88 with the -A and -C switches. The syntax for merge is:

```
    merge <file>
```

where <file> is the name of the .C and .A files to be merged. The output of the merge program is placed in the file <file>.L For example, to get a merge listing of a file named BLIP:

```
    c88 BLIP A C
    merge BLIP
```

The file BLIP.L will be created with the merged information.

MORE: a file screen listing utility.

more is a program used to view text files. This version is slightly more sophisticated than the program which is delivered with PC-DOS. It includes the ability to search for a given string. The syntax for more is:

```
    more file ...
```

where file may contain wildcard characters.

When the program pauses and displays the `--MORE--` prompt, you have a number of choices:

`/<pattern>`	searches for a line with the given pattern. If the pattern is not found, the --MORE-- prompt is redisplayed.
Q	terminates the program.
N	finds the next line with the same pattern used in the last / command.

Any other character will display the next 22 lines of the file.

PCmake : a program maintenance utility

PCmake is loosely based on the UNIX *make* program. This program is used to keep track of depedencies between files and, based on the last modification date, ca generate a batch (.BAT) file for updating out-of-date files. This batch file can then be executed to perform the update.

Invocation Syntax:

```
    PCmake [<dependent>] [-f<input>] [-o<output>][-d][-a][-i]
```

`<dependent>`	is the name of the dependent file to check for update requirements (the first dependent filename used in the input file is used by default).
`-f<input>`	causes PCmake to get its input from the given file ("makefile" is used by default).
`-o<output>`	causes PCmake to write the batch file to the indicated file ("makeit.bat" is used by default).
`-d`	lists the dependency tree display to the console.
`-a`	insures that all files are updated regardless of the modificaton dates.
`-i`	turns off the error checking produced in the .bat file

The input file to PCmake has the following syntax:

```
    <dependent> [<dependencies>] <constructor>
```

`<dependent>`	is the file which is to be updated as necessary.
`<dependencies>`	is a list of filenames on which the file is dependent. This list can span multiple lines and is terminated by the "]".
`<constructor>`	is a number of lines, terminated by a blank line, which are executed in order to update the file.

The simplest example of a file dependency is an object file's dependency on the corresponding source file. When the source file is altered, PCmake can be used to detect that the object file is "older" than the source file and can generate the compile command to update the object file. For example, if we have a source file named `foo.c`, with its corresponding object file `foo.o` and executable file `foo.exe`, the makefile might look as follows:

```
    foo.exe [ foo.o cstdio.s ]  bind foo.o
    foo.o   [ foo.c stdio.h ]   c88 foo.c
```

Note the "extra" dependencies of the .EXE file on the c library and the object file dependecy on the standard include header file. If a modification was made to foo.c and PCmake subsequently executed, it would produce a batch file with the following lines:

```
    c88 foo.c
    IF ERRORLEVEL 1 GOTO stop
    bind foo.o
    IF ERRORLEVEL 1 GOTO stop
    :stop
```

which would result in updating the file foo.exe. However, if the source file had not been changed, but a new cstdio.s library was copied onto the disk, then PCmake would simply generate:

```
    bind foo.o
    IF ERRORLEVEL 1 GOTO stop
    :stop
```

since the source file doesn't need to be recompiled.

MACROS

A macro is defined by beginning a line with the macro character '$' followed by the macro name and the definition. For example:

```
    $FOO a.o b.o c.o
```

would define the macro `$FOO` as the string `a.o b.o c.o` and could be used in the dependency line

```
    xx.exe [ a.o b.o c.o ]
```

as

```
    xx.exe [ $FOO ]
```

Macros may also be used in the constructor lines to allow fast substition for different environments. For example: to enable the checkout flag on the compiler while you are debugging, add the macros

```
    $C88 c88
    $C88FLAGS -c
```

and construct your dependency lines like

```
    foo.o    [ foo.c stdio.h ] $C88 foo.c $C88FLAGS
```

When you want to generate a production version, redefine `$C88FLAGS` to null by specifying

```
    $C88FLAGS
```

and then use the -a option to recompile everything without the checkout option.

SHORTCUTS

PCmake also recognizes certain filename extensions and can produce dependency and constructor lines without further user input. The .exe extension is recognized as being dependent on a file with the .o extension and is created by using the binder. The .o extension is recognize as being dependent on either a .c or .a file (in that order) depending on which file exists and is created by either compiling or assembling the file.

For a very simple makefile, with only a single file that must be compiled (foo.c) and bound, we can simply use the following two lines:

```
    foo.exe
    foo.o
```

PCmake will automatically generate the dependency on foo.o for the foo.exe line and the constructor line

```
    $BIND88 foo.o $BIND88FLAGS
```

For the second line, PCmake will find the file foo.c on the disk and make the dependency and constructor line

```
    $C88 foo $C88FLAGS
```

Note the use of the macros `$BIND88 $BIND88FLAGS $C88 $C88FLAGS`. This allows you to tailor the defaults to your system. For example, if your compiler is in the root directory and on the default directory, you can add the macro definition

```
    $C88 /c88
```

to your makefile and the PCmake program will replace the macro name $C88 with the string /c88.

The defaults for the internal macros are:

```
    $BIND88        bind88
    $BIND88FLAGS   <none>
    $C88           c88
    $C88FLAGS      <none>
    $ASM88         asm88
    $ASM88FLAGS    <none>
```

Minimizing the first example would yield the following makefile:

```
    foo.exe [ foo.o cstdio.s ]

    foo.o
```

This will generate the same .BAT file as the original example. Note the blank line between the two lines. This is required whenever there is a dependency list to terminate the list of constructor lines.

PROFILE: a performance monitor utility

Profile is a performance monitoring tool for use with the C88 compiler. It provides a statistical measure of the amount of time spent in a program or procedure within the program.

With the version 2.3 or later compiler, specifying the check option (-C) for both C88 and BIND will create a .CHK file. The profiler uses the .CHK file to produce symbolic output instead of the standard hexadecimal output.

Profile only works on the IBM-PC and very similar machines as it manipulates the hardware timers. It also requires the use of MS-DOS V2.xx or later.

To invoke the profiler, type:

```
    A>profile
```

The profiler will load and request the command line of the program to be analyzed. Enter the command line as if you were invoking the program normally. The profiler will then display one of the following two menus:

If a corresponding .CHK file exists:

```
All  List-procs  Procedure  Range  Quit  Start
```

or if no .CHK file exists:

```
Range  Quit  Start
```

Make a menu selection by typing the first character of the appropriate menu item.

All indicates that the entire program is to be monitored and broken down by procedure.

List-procs — displays the procedure names and addresses. When entering the name of the procedure, the wildcard characters * and ? may be used. * will match anything, ? will match any single character. All names which match the pattern will be displayed.

Procedure — indicates that a single procedure is to be monitored. The output will be displayed with the line numbers within the procedure.

Range — indicates that a specific range of addresses within the program is to be monitored.

Quit aborts the current profiling session.

Start begins the execution of the program.

After the monitored program exits, control is returned to the profiler which will display the execution histogram and the following menu:

```
Disk-list  List-again  Quit
```

Disk-list — indicates that the profiling histogram should be written to a disk file. The profiler will prompt for the name of the file.

List-again — indicates that the histogram should be redisplayed from the beginning.

Quit exits the profiler.

Use the space bar to display the next set of procedures or line numbers in the histogram. The histogram includes the entries "system" and "other". "system" is the amount of time measured outside of the executing program's code segment. "other" is the amount of time spent within your code segment but outside of the measured range.

The profiler also uses two other programs, profstar.exe and profend.exe. These programs may be placed in the current directory or in a directory identified by the PATH environment variable.

Sampling Algorithm

Internally, the profiler maintains 1024 counters which are used to monitor the activity within certain regions of memory. The location and size of these regions depend on the *range* specification. The size of each region is determined by dividing the entire range into 1024 equal size pieces. The minimum size of a given piece is 1 byte. For example, if the selected range is 0x1C to 0x401C, the size of each region is 16 bytes. Each time the timer interrupt is generated, the counter associated with the location of the instruction pointer is incremented. In this example, an IP value between 0x1C and 0x2C will appear in the first region. You can see that the selection range sets the granularity of the sampling mechanism. Shorter ranges lead to finer granularity and therefore more accurate measurements. Because of the granular nature of the sampling method, some sampling errors may occur. If the end of one procedure and the beginning of another procedure happen to fall into the same sampling region, then the second procedure will inherit the count from the end of the first procedure.

RM: a file removal utility

rm is a simple program to delete files. The main difference between rm and DEL is that rm will accept multiple filenames and can work interactively. The syntax for rm is:

```
rm [-i ] [-l] filename ...
```

`-i`	sets the interactive mode. rm will ask for confirmation on each of the files before deleting it.
`-l`	sets the list mode to list the names of the files as they are being deleted.
`filename`	may contain the wildcard characters * and ?

If no filenames are given, the program will display its syntax description. (e.g. if you only type rm or if there are no files which match) . rm does not support deletion of directories.

SENSE87: an 8087/80287 SENSING LIBRARY

SENSE87 was developed by Dan Lewis, Key Software Products, 440 Ninth Avenue Menlo Park, CA 94025 (415) 364-9847

SENSE87.S contains everything you need to make an 8087/80287 sensing library for your C88 compiler. This effectively eliminates the need to build two different versions of your programs, one for machines that have an 8087/80287 coprocessor, and another for those that don't. Most people have been taking the easy way out, creating code that never uses the coprocessor, even if one is installed; now your program can automatically sense the absence or presence of a coprocessor, and take advantage of its speed if installed.

The modules included here were created by combining 8087 and non-8087 routine with 8087- sensing software that automatically chooses between the original routines. Since both the coprocessor and non-coprocessor versions must be in the sensing library, your EXE file size (in particular, the code segment) will increase, probably by about 2K bytes, but will depend on how many floating point functions your program pulls in from the library.

NOTE: If you are using the O88 optimizer from Key Software Products, be sure to disable the 8087 option (-7) so that the CALLs to the floating-point library are NOT replaced by in-line 8087 instructions!

HOW TO CREATE YOUR SENSING LIBRARY

The file SENSE87.S contains the object files required to create an 8087-sensing library CSTDIO.S from the standard 8087 library (CSTDIO7.S). Large Case files are BSENSE87.S, BCSTDIO.S and BCSTDIO7.S. The file SENSE87.BAT (BSENSE87.BAT) will create the sensing library in the directory you specify.

For hard disks, with a copy of distribution disk #1 in drive A, enter:

```
C>a:sense87 a:
```

For floppy disks:

```
B>a:sense87 b:
```

TECHNICAL DETAILS FOR THOSE WITH INQUIRING MINDS

The modified library entry points are the following: (All other library routines that do floating point computations do so by calls to this floating point kernel.)

_chk8087 (added)	_testinit (deleted)	_floadd
_floade	_floadl	_fstored
_fstoree	_fstorel	_fxch
_fclear	_fcmpkeep	_fcmp
_fneg	_fnot	_fzero
_fis	_fdec	_finc
_fsub	_fadd	_fdiv
_fmul	sqrt_	floor_
ceil_	log_	exp_
atan_	_tan_	

The replacement routines each begin with a JMP instruction that jumps through a data segment "vector". The vector initially points to a short setup routine which checks for the 8087/80287. The first execution of the setup routine replaces the vector with a pointer to the appropriate version of the code, and then jumps through the updated vector.

The 8087/80287 check is handled by a public routine called "_chk8087" which actually only checks for the coprocessor once, and saves the result so it can be returned without resetting the coprocessor on subsequent calls made by other routines.

The technique used to sense the coprocessor is to execute an FNINIT followed by a FNSTCW, then to examine the most significant byte of the control word stored by the FNSTCW for the appropriate value. Some late-model PC/AT's are known to destroy the segment containing the location of the stored control word, starting from its offset to the end of the segment. To avoid this problem, the routine copies DS to ES and the offset of the destination address into SI, then backs off ES while incrementing SI until the offset is within the range FFF0-FFFF. If it turns out to be FFFF, then it is changed to FFF0 and ES is incremented. Of course this requires that 32 bytes be reserved within the data segment where the control word is to be stored.

TOOLBOX.S : a library of useful routines

The following routines are in `TOOLBOX.S` located on Disk #1. To include them ir your program, include `TOOLBOX.S` in your `BIND` command line

```
    BIND ... TOOLBOX.S ...
```

FINDFILE

```
int findfile(filename, target_buf)
```

searches for the file given in filename by checking the current directory and the directories listed in the PATH environment variable. If the file is found, findfile returns 1 and the target_buf area contains the FULL pathname. If the file isn't found, 0 is returned.

Requirements: This routine uses 300 bytes of stack.

Limits: only 256 bytes of the PATH variable will be searched.

LINE INPUT ROUTINES:

```
    char *cur_line;
    int  line_number;
    line_start(fname, position, first_line)
```

opens the file fname at the location position (long). The first line of the file will be pointed to by cur_line after this call and line_number will be first_line. Returns 1 if successful, 0 otherwise.

```
    line_next()
```

Returns 1 if another line is available, sets cur_line to point at the new line and increments line_number; returns 0 if no more lines are available.

```
    line_stop()
```

closes the input file.

```
    line2_start, line2_next, line2_stop, cur2_line,
    line2_number
```

Same as above for a second file.

WILDCARD EXPANSION

```
main(argc, argv)
```

This is a C main procedure which expands wildcard filenames out to multiple arguments. Command line items such as *.c return all the .c files in the current directory. The user main program must be named main1 instead of main for the linkage to work.

Chapter 10

The CSTDIO Library

Introduction	10.1
Names	10.1
Program Initialization	10.2
Calling Conventions	10.4
Memory Management	10.9
Input/Output Library	10.11
Directory Level Functions	10.11
File Level Functions	10.11
Stream Level Functions	10.12
Handle Level Functions	10.13
Screen Level Functions	10.13
Console Level Functions	10.14
Math Library	10.15
System Interface	10.16
Environment	10.18-1
Library	10.18-2
Headers	
assert.h	10.18-2
ctype.h	10.18-2
math.h	10.18-3
setjmp.h	10.18-3
stdarg.h	10.18-3
stdio.h	10.18-4
stdlib.h	10.18-5
string.h	10.18-6
Functions & Macros	
Alphabetical by name	

Introduction

This section describes the standard library, CSTDIO.S, for the C88 C compiler and ASM88 assembler. This library includes routines similar to routines available in UNIX with some inevitable differences due to the DOS Operating System.

All the routines are in the CSTDIO.S file provided on the distribution disk. For BIND to execute correctly, this file must be either on the default drive/directory, in a directory listed in the PATH system parameter, or on the drive/directory referred to by the -L option.

Ther CSTDIO7.S library has the same functions as CSTDIO.S, but requires an 8087 math coprocessor to perform floating-point operations. To use the 8087 library, rename CSTDIO7.S to CSTDIO.S.

Names

Public names starting with the underline character ('_') are used by C88 internal routines and should be avoided. Names of this form are also used for user-callable routines such as `_move()` that have names that might conflict with user names.

C88 automatically appends the underline character ('_') to public names to avoid conflicts with assembly language reserved words. ASM88 does not do this so the underline must be manually appended to publics used to link with code generated by C88. For example, the C `puts()` routine should be referred to as `puts_` from assembler. Unlike UNIX, BIND ignores the case of publics, so `puts_` matches `PutS_`.

Program Initialization

BIND inserts a `jmp _csetup` as the first executable instruction in the program. _CSETUP performs the following initialization functions:

	Memory Model	Action
1.	Small Case	Sets the data/stack segment size to the lesser of: the amount available memory, 64K, or the size of the static data area plus the BIND -S option,
	Large Case	Sets the stack size to the value specified in the BIND -S option, otherwise sets the stack size to 8K.
2.	Both	Formats `argc` and `argv[]` from the Program Segment Prefix,
3.	Both	Zeros the Uninitialized Data Area, and
4.	Both	Calls `main(argc, argv)`

The figure below shows the Small Case memory layout after initialization:

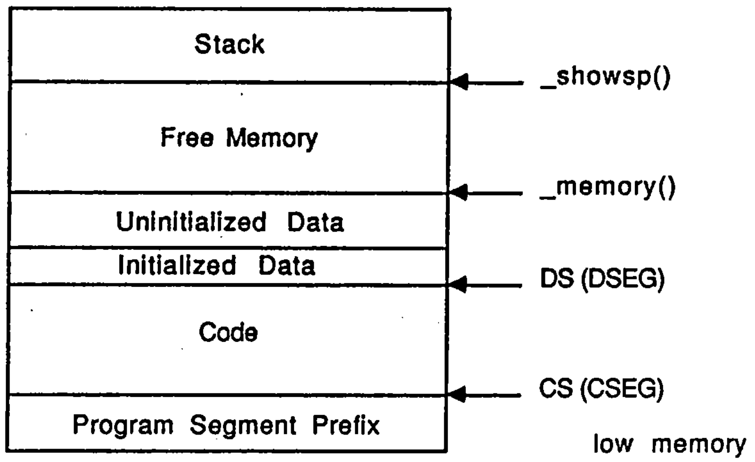


Figure 10-1
Small Case memory model

The figure below shows the Large Case memory layout after initialization:

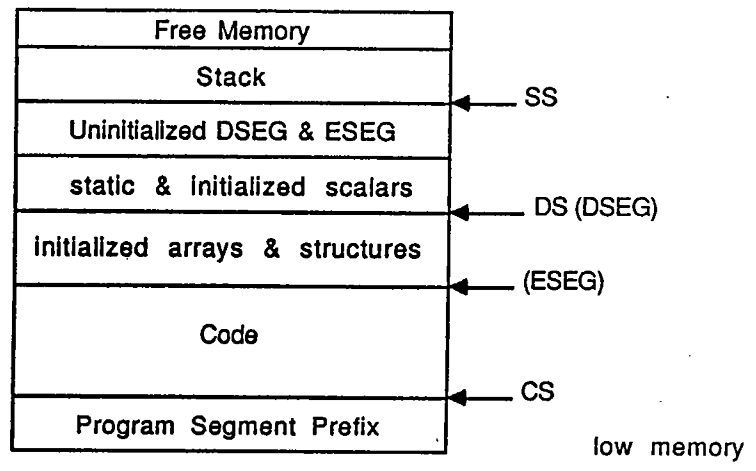

Free Memory
Stack
SS
Uninitialized DSEG & ESEG
static & initialized scalars
DS (DSEG)
initialized arrays & structures
(ESEG)
Code
CS
Program Segment Prefix
low memory

Figure 10-2
Large Case memory model

The initialization code saves the address of the Program Segment Prefix. To access the PSP address from C use

```
        extern char * _pcb;
```

From assembly language use

```
        dseg
        public _pcb_:word
```

Assembly language main programs that require normal C address space initialization should contain the following:

```
        PUBLIC MAIN_
    MAIN_:
```

See the Memory Management discussion below for information on how to access the free memory.

The -A option of BIND inhibits the call to `_csetup`. Execution starts with the first instruction of the first filename specified to BIND. See the file BUF128.A on your distribution disk for an example.

On entry, the registers have the following values:

`CS`	Address of Code Segment. Execution starts at `CS:0`.
`SS`	Address of Data Segment.
`ES,DS`	Address of Program Segment Prefix
`SP`	Stack size set by BIND

The library module that contains `_csetup` also contains the following functions — thus they cannot be replaced in CSTDIO.S without removing `_csetup`.

```
ci()       co()       csts()     exit()
getchar()  putchar()  puts()     _memory()
_setsp()   _showcs()  _showds()  _showsp()
```

Calling Conventions

For a given C function, the stack is arranged as follows:

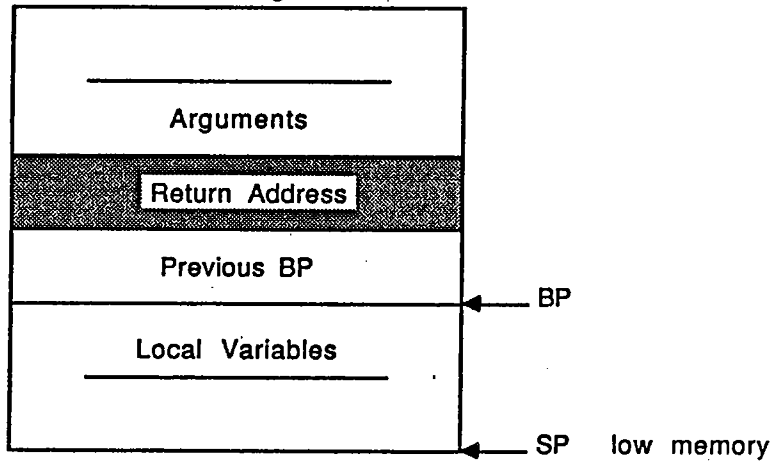


Figure 10-3
Stack Frame Layout

The memory model layout is defined in assembler as

```
        if      LARGE_CASE

ARGS_   equ     bp+6
SARG_   equ     bp+10
CALL_   lcall
RET_    lret

        else

ARGS_   equ     bp+4
SARG_   equ     bp+6
CALL_   call
RET_    ret

        endif
```

Called functions must preserve `CS`, `DS`, `SS`, `SP`, and `BP` across the function call. No other registers have to be preserved.

Function arguments are pushed on to the stack, rightmost argument first. The calling function cleans up the stack. For example

```
int *i;
zip(i, 6);
```

would generate the following code in the Small Case memory model

```
mov    ax,6
push   ax
push   word i_
public zip_
CALL_  zip_
add    sp,4
```

and would generate the following code in the Large Case memory model

```
mov    ax,6
push   ax
push   word i_[2]
push   word i_
public zip_
CALL_  zip_
add    sp,6
```

The `word` modifier is required because C88 allocates variables in bytes rather than words, double-words, ... The `add sp,` removes the words that were pushed as parameters to `zip_`. Note that C88 appended '_' to names. If there had been no local variables defined in the calling function, the clean-up code would have been

```
        mov   sp,bp
```

which is faster.

Data is pushed on the stack as follows:

`char` pushed as a word, with high-order byte set to zero

```
     mov  AL,data_
     mov  AH,0
     push AX
```

`int`
`unsigned` pushed as a word

```
     push WORD data_
```

`long` pushed as two words, with least-significant word pushed last

```
     push WORD data_[2]
     push WORD data_[0]
```

`float` Changed to `double` and pushed with least-significant word pushed last

```
     mov  si,offset data_
     mov  ax,ds        ; Large Case only
     mov  es,ax        ; es:si -> float
     PUBLIC _FLOADE    ; load float
     CALL_ _FLOADE
     PUBLIC _FPUSH     ; push double
     CALL_ _FPUSH
```

`double` pushed as four words with least-significant word pushed last

```
     push WORD data_[6]
     push WORD data_[4]
     push WORD data_[2]
     push WORD data_
```

`struct` push `(sizeof(struct) + 1) >> 1` words, with least-significant word pushed last.

```
          mov  cx,nn     ; size in words
          sub  sp,cx     ; make room on stack
          mov  di,sp     ; target
          mov  si,offset data_  ; source
          ; set up ds for memory model
          mov  ax,ss     ; setup
          mov  es,ax     ;    es
          cld            ; set direction up
     rep  movsw          ; copy to stack
```

`*` Small Case memory model — 2-byte pointer

```
          push WORD pointer_
```

Large Case memory model — 4-byte pointer

```
          push WORD pointer_[2]
          push WORD pointer_
```

The usual preamble for a called function is

```
          PUBLIC fname_
     fname_:
          push bp        ; save old frame pointer
          mov  bp,sp     ; establish local frame
```

For functions that don't return structures, parameters begin in the local frame at `[ARGS_]`, and continue upward based on the size of each parameter. Thus for the fragment

```
     blip(x, y, z)
     int x;
     long y;
     double z;
```

the parameters would be referenced in Assembler as

```
     mov  cx,WORD [ARGS_]     ; x_
     mov  ax,WORD [ARGS_+2]   ; lsw of y_
     mov  dx,WORD [ARGS_+4]   ; msw of y_
     lea  si,[ARGS_+6]        ; addr of z_
```

For functions that return structures, `[ARGS_]` contains a pointer to where the structure should be returned, and the arguments begin at [SARG_]. So if the above fragment was

```
struct foo blip(x, y, z)
```

the parameters would be

```
mov  cx,WORD [SARG_]     ; x_
mov  ax,WORD [SARG_+2]   ; lsw of y_
mov  dx,WORD [SARG_+4]   ; msw of y_
lea  si,[SARG_+6]        ; addr of z_ SS:SI
```

Local variables are allocated below the current frame pointer regardless of the memory model or what the function returns, so that the fragment

```
{
int aa[2];
long b;
```

would be referenced as

```
sub  sp,8         ; allocate space for locals
mov  ax,[bp-4]    ; aa_[1]
mov  dx,[bp-8]    ; msw b_
```

The standard exit sequence is

```
mov  sp,bp    ; reclaim any local space
pop  bp       ; old frame pointer
RET_          ; caller will clean up stack
```

Values are returned from functions according to the following table

`char` `int` `unsigned`	returned in `AX`. `char` values are returned in `AL` with `AH` set to zero
`long`	returned in `DX:AX`. (AX contains lsw)
`double` `float`	returned on floating point stack (s/w or 8087).

`struct`	returned to address in `[ARGS_]`
`*`	Small Case memory model — returned in AX Large Case memory model — returned in ES:SI.

Memory Management

The Memory management functions are:

Function	Effect
`calloc()`	Allocates a block of data and clears it to zeroes
`free()`	Marks a block of data as available for allocation
`freeall()`	Initializes the free memory area
`malloc()`	Allocates a block of data
`realloc()`	Resizes an existing allocated block of data

The Small Case memory model data segment looks like

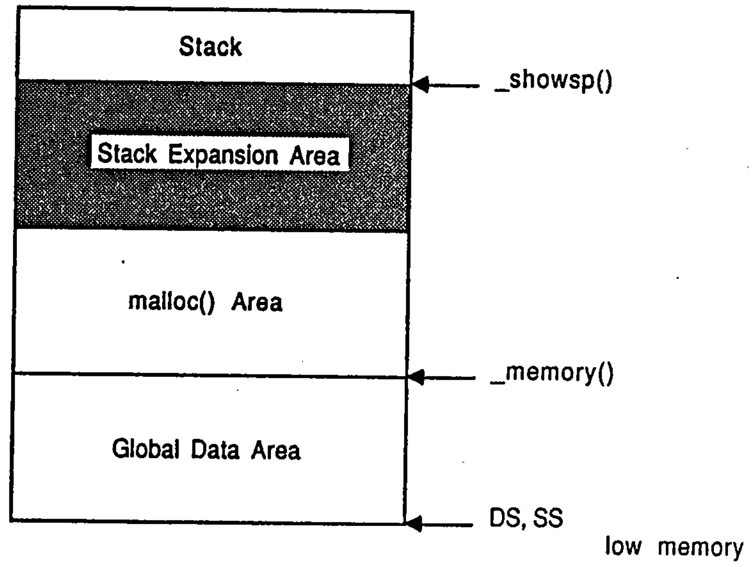


Figure 10-4
Small Case memory management

The default *stack expansion area* is 1024 bytes. The size of the area is fixed by `freeall()`, specifying the size of the area. Note that `freeall()` releases all allocated storage in the *malloc()* area, so in general it is best to call `freeall()` prior to any `malloc()` or `calloc()` calls.

The *malloc()* area is divided into blocks with the following format:

```
struct {
     char status;
     unsigned size;
     char data[1];
     };
```

status is one of: allocated (`0xAB`), unallocated(`0x9D`), or end-of-area marker (0x`C6`). *size* is the size of *data* in bytes. The address of *data* is returned by *malloc*, *calloc*, and *realloc*, and used by *free*.

The following function prints out a map of the memory allocation area.

```
#define UNALL 0x9D
#define ALLOC 0xAB
#define EOA   0xC6

printMap(){
     char *cp, *_memory();
     struct {
         char status;
         unsigned size;
         char data[1];
         };

         cp = _memory();
         while(cp->status != EOA) {
             printf("%5u %2salloc bytes at %u\n",
                 cp->size,
                 cp->status == ALLOC ? "" : "un",
                 cp);
             cp = cp->data + cp->size;
             }
         }
```

free marks a block as unallocated. *malloc* searches the allocation area in order from bottom to top.

Thus in the following fragment

```
          fp = malloc(size);
          free(fp);
          np = malloc(size);
```

fp may not equal *np* .

In the Large Case memory model, `malloc()`, `calloc()`, `realloc()`, and `free()` use the DOS memory management facilities.

Input/Output Library

The I/O routines work at different levels. Directory level functions manage the DOS 2 and later directory structure. File level functions manage the contents of directories. Stream level function access files as a sequence of bytes. Handle level function manage files as blocks of data.

Screen level functions simplify the interface between C programs and the IBM-PC and its clones. Console level functions read and write to the console.

Directory Level Functions.

Directory level functions manage the DOS 2 and later directory structure. Each of the function works on a directory specified by a pathname.

Function	Effect
`chdir()`	Changes the current working directory
`getdir()`	Returns the pathname of the current working directory
`mkdir()`	Creates a new directory
`rmdir()`	Deletes an existing directory

File Level Functions

File level functions manage the contents of directories. Each function works on a file identified by a pathname or file handle.

Function	Effect
`chmod()`	Changes the file attributes
`filelength()`	Returns the length of the indicated file

`isatty()`	Tests if the file is a character device
`locking()`	Locks parts of a file (MSDOS 3.0 and later)
`rename()`	Changes the name of a file
`unlink()`	Deletes an existing file

Stream Level Functions

Stream level function access files as a sequence of bytes. They buffer the data read from and written to disk files in private storage areas. They use a pointer to a FILE (defined in `stdio.h`) to associate a stream with the pathname of the file.

Function	Effect
`fclose()`	Closes a stream
`fflush()`	Writes buffered data out to disk file
`fgetc()`	Reads a character from a stream
`fgets()`	Reads a string from a stream
`fileno()`	Returns the file handle of a stream
`fopen()`	Opens a stream
`fprintf()`	Writes formatted string to stream
`fputc()`	Writes a character to a stream
`fputs()`	Writes a string to a stream
`fread()`	Reads a block of data from a stream
`freopen()`	Redirects a FILE pointer
`fscanf()`	Reads formatted data from a stream
`fseek()`	Positions a stream at a specific character
`ftell()`	Returns the position of a stream
`fwrite()`	Writes a block of data to a stream
`getc()`	Reads a character from a stream
`getchar()`	Reads a character from *stdin*
`getw()`	Reads an `int` from a stream
`printf()`	Writes formatted string to *stdout*
`putc()`	Writes a character to a stream
`putchar()`	Writes a character to *stdout*
`puts()`	Writes a string to a stream
`putw()`	Writes an `int` to a stream
`rewind()`	Positions a stream to the beginning of the file
`scanf()`	Reads formatted data from *stdin*
`sprintf()`	Writes formatted string to a string
`sscanf()`	Reads formatted data from a string
`ungetc()`	Pushes one character back into a stream

There are five predefined streams available for reading or writing. They don't have to be opened before, or closed after use. To refer to them, use the following predefined FILE pointers (defined in `stdio.h`):

Stream	Device
stdin	keyboard, can be redirected
stdout	display, can be redirected
stderr	display, can not be redirected
stdaux	COM1
stdptr	LPT1

Handle Level Functions

Handle level functions manage files as blocks of data. They do not buffer or format the data. They use an `int` to associate a file *handle* with a pathname.

Function	Effect
`close()`	Closes a file
`creat()`	Creates and opens a file
`dup()`	Makes another *handle* for a file
`dup2()`	Redirects a *handle*
`lseek()`	Positions a file at a given location
`open()`	Opens a file
`read()`	Reads a block of data from a file
`write()`	Writes a block of data to a file

The maximum number of handles that can be open at one time is either 20, or the number specified in CONFIG.SYS, whichever is less. See Installing the Software section of Chapter 2 for details about CONFIG.SYS.

Screen Level Functions

Screen level functions simplify the interface between C programs and the IBM-PC and its clones. These routines are not in the standard CSTDIO.S library but are distributed in source form in the file PCIO.A. To use these routines, they must be assembled and bound in. For example:

```
A>asm88 b:pcio
A>bind b:blip b:pcio
```

See the comments in the IBM Technical Reference Manual for details on the BIOS interface used by PCIO.

See the LIB88 chapter for details on installing PCIO.O in CSTDIO.S.

Function	Effect
`scr_aputs()`	Writes a string to the screen with a specified attribute
`scr_ci()`	Reads a character from the keyboard
`scr_co()`	Writes a character to the display
`scr_csts()`	Tests for the availability of keyboard data
`scr_clr()`	Erases the entire screen
`scr_clrl()`	Erases from the cursor to end-of-line
`scr_cls()`	Erases from the cursor to end-of-screen
`scr_cursoff()`	Turns the cursor off
`scr_curson()`	Turns the cursor on
`scr_rowcol()`	Moves the cursor to the specified row and column
`scr_scdn()`	Scrolls the screen down 1 line, starting at line 3
`scr_scrdn()`	Scrolls an area of the screen down 1 line
`scr_scrup()`	Scrolls an area of the screen up 1 line
`scr_scup()`	Scrolls the screen up 1 line, saving lines 1 and 2
`scr_setmode()`	Change the mode of the CGA
`scr_setup()`	Initialize the screen level functions
`scr_sinp()`	Reads the character at the current cursor location

Console Level Functions

Console level functions read and write to the console. They may be redirected.

Function	Effect
`ci()`	Reads a character from the keyboard, no echo
`co()`	Writes a character to the display
`csts()`	Tests for keyboard input

Math Library

If any of the transcendental or sqrt() functions are used, *include* the file `math.h` or the equivalent declarations to specify them as returning a `double`.

Function	Effect
`abs()`	Absolute value of `int`
`acos()`	Arc-cosine of radian argument
`asin()`	Arc-sine of radian argument
`atan()`	Arc-tangent of radian argument
`ceil()`	Returns ceiling of its argument
`cos()`	Cosine of radian argument
`exp()`	Exponential function
`fabs()`	Absolute value of `double`
`floor()`	Returns floor of its argument
`frexp()`	Disassembles a `double`
`labs()`	Absolute value of `long`
`ldexp()`	Assembles a `double`
`log()`	Log function
`modf()`	Decomposes a `double`
`pow()`	Power function
`sin()`	Sine of radian argument
`tan()`	Tangent of radian argument

`math.h` includes the statement

```
    extern int errno;
```

`errno` is set to a non-zero value when: a floating point stack errors, an argument to a math routine is out of range, or the result of a math routine would under/overflow. Error codes and names (defined in `math.h`) are:

30 ESTK — F/P stack overflow. The most probable cause is calling a function that returns a `double` without declaring it as such to the compiler. After eight calls, the f/p stack will be full.

33 EDOM — invalid argument, i.e., sqrt(-1.0).

34 ERANGE — result would under/overflow, i.e., tan(PI/2.0).

The function `rerrno()` is called by the floating point routines whenever an error is detected. `rerrno()` prints out an appropriate error message and calls `exit()`. In order to bypass this effect, install the following function in your program

```
    rerrno() {;}  /* null function to suppress printing */
```

SYSTEM INTERFACE

The System Interface provides access to low-level DOS and BIOS functions.

Function	Effect
`chain()`	Transfers control to another .EXE file, no return
`exec()`	Transfers control to another .EXE file
`_doint()`	Invokes a 8088 interrupt
`_os()`	Invokes simple DOS interrupt (21H)

chain and *exec* will load and execute an arbitrary program. *exec* returns control to your program, *chain* does not. You specify the complete pathname of the program (including the .EXE or .COM suffix) and the arguments to the program. *chain* and *exec* are in the EXEC.O file provided on the distribution disks.

exec will return the completion code from the program or -1 if an error occurred loading the program. Completion codes are set for programs running under DOS 2.0 or later versions of the operating system. If a program exits with

```
        exit(n);
```

the system ERRORLEVEL will be set to n. A program that returns from the `main` function other than by `exit()` sets ERRORLEVEL to zero. ERRORLEVEL can be tested with the DOS batch file IF command. See the section under 'BATCH' in the DOS manual for details on the IF command.

To invoke a Batch file, or a DOS built-in command, use COMMAND.COM with the '/c' switch as follows:

```
        char shell[128];

        getenv("COMSPEC", shell);
        exec(shell, "/cxxx");
```

where *xxx* is one of the DOS built-in commands ('dir', 'copy', ...) or the name of a batch file, including the trailing .BAT.

```
        exec(shell, "/cc:\\autoexec.bat");
```

Remember that two backslashes are required to insert a single backslash in a string.

Invoking COMMAND.COM with no parameters will start another DOS shell (like F9 in SEE). To return, enter at the command prompt

```
        exit
```

C88 normally allocates a stack as large as possible. This is not desirable when using *exec* , as little memory may be left for the second program. The -Shhhh option of the BIND program should be used to reduce the size of the stack and consequently the size of the program. Remember that the hhhh value of the option is in hex and that it must be large enough for all parameters and locals active at one time. An extra 0x100 (256) bytes should also be added for any system calls.

chain loads the new program physically above itself in memory so stack size is irrelevant. *chain* is contained in the EXEC.O file on your distribution disk. When using *chain* , EXEC.O should be the first parameter to BIND

```
        BIND EXEC progName -oprogName
```

One way to pass data via *exec* to another program is to pass a pointer to data block. The driver program for a menu program could be:

```
        struct data {
             int anydata;
                  . . .
             char nextMenu[13];
             } comArea;

        main(){
             char parms[10];

             strcpy(comArea.nextMenu, "MENU1.EXE");
             sprintf(parms, "%u %u", _showds(), &comArea);
             while(exec(comArea.nexMenu, parms) > 0)
                  ;
             }
```

Each menu program copies `comArea`, as follows:

```
struct data {
     int anydata;
          . . .
     char nextMenu[13];
     } comArea;

main(argc, argv)
char *argv[];{
     unsigned seg, off; /* driver comArea */

     sscanf(argv[1], "%u %u", &seg, &off);
     _lmove(sizeof(struct data),
           off, seg, &comArea, _showds());
     /* process menu */
     strcpy(comArea.nextMenu, "MENU2.EXE);
     _lmove(sizeof(struct data),
           &comArea, _showds(), off, seg);
     exit(1); /* zero return terminates driver */
     }
```

_doint will cause software interrupt *inum* and may be used to call whatever routines are available in the particular machine. The values of the registers can be specified before, and read after, the call to *_doint*.

_os provides an elementary interface to the BIOS. *inum* goes into AH and *arg* into DX, and an `int 21H` is executed.

ENVIRONMENT

The function called by the startup code is named `main`. There is no predefined prototype for this function. It can be defined with no parameters:

```
    int main(void) {
```

or with two parameters (refered to here as argc and argv, though any names may be used, as they are local to the function in which they are declared):

```
    int main(unsigned argc, char * argv[]) {
```

or with three parameters:

```
    int main(unsigned argc, char * argv[], char * envp) {
```

If they are defined, the paramters of `main` have the following characteristics:

- `argc` is non-zero,
- `argv[argc]` is a null pointer,
- `argv[0]` through `argv[argc-1]` contain pointers to strings, which are portions of the command line arguments used to invoke the program. The strings consist of character sequences that do not contain whitespace,
- under DOS 3, `argv[0]` conatins a pointer to the name that was used to invoke the program. Under DOS 1 and DOS 2, `argv[0]` is a null pointer,
- `envp` is a pointer to the DOS environment string. In small-case, `envp` is the segment number of the environment string — use `_peek()` or `_lmove()` to access the string. In large-case, `envp` is a pointer to the string.

In addition, the following envrionmental variables are available:

- `extern unsigned environ` contains the paragraph number of the environment string
- `extern unsigned _version` contains the current DOS version, with the major release number in the low-order byte, and the minor number in the high-order byte.

- `extern char _osmajor` contains the major DOS release number.
- `extern char _osminor` contains the minor DOS release number.
- `extern unsigned _psp` contains the paragraph number of the Program Segment Prefix (PSP).

LIBRARY

Each library function is declared in a header, whose contents are made available via the `#include` preprocessing directive. The header declares a set of related functions, plus any necessary types and additional macros needed to facilitate their use. All external identifiers declared in any of the headers are reserved, whether or not the associated header is included. All external identifiers and macro names that begin with an underscore are also reserved.

The following section describes the headers, and their associated functions, in a general way. Following that is a description of each function, arranged alphabetically.

Headers

`<assert.h>` This header defines the `assert` macro and refers to another macro, `NDEBUG`, which is *not* defined by `<assert.h>`. If `NDEBUG` is defined as a macro name at the point in the source file where `<assert.h>` is included, the macro has no replacement text, as in

```
# if defined NDEBUG
# define assert(expr)
# endif
```

`<ctype.h>` This header defines several macro implementations of character testing and mapping functions. The macros defined, which are also available as functions, are:

```
isalnum(int) isgraph(int) ispunct(int) isxdigit(int)
isalpha(int) islower(int) isspace(int) tolower(int)
iscntrl(int) isprint(int) isupper(int) toupper(int)
isdigit(int)
```

`<math.h>` This header declares several mathematical functions and defines three macros. The functions take `double` arguments and return `double` values.

The macros defined are `EDOM`, `ERANGE`, and `HUGE_VAL`.

The trignometric functions are:

```
acos()      atan()      cos()       tan()
asin()      atan2()     sin()
```

The exponential and logarithmic functiona are:

```
exp()       ldexp()     log10()     frexp()
log()       modf()
```

The power functions are: `pow()` `sqrt()`

The miscellaneous functions are

```
ceil()      fabs()      floor()
```

`<setjmp.h>` This header declares two functions and one type, for bypassing the normal function call and return discipline.

The type declared is `jmp_buf` which is an array type used to restore a calling environment.

The functions are: `longjmp()` `setjmp()`

`<stdarg.h>` This header declares a type and defines three macros, for advancing through a list of arguments whose number and types are not known at compile time.

The type declared is `va_list` which is used to accessing the arguments.

The macros are: `va_start()` `va_arg()` `va_end()`

`<stdio.h>` This header declares two types, several macros, and many functions for performing input and output.

The types are `FILE` which is an object describing a stream, and `fpos_t` which is an object describing a location within a file.

The macros are:

```
EOF          SEEK_CUR     stdout      clearerr
ERR          SEEK_END     stderr      feof
FALSE        SEEK_SET     stdaux      ferror
NULL         stdin        stdprn
```

The file operation functions are:

```
remove()     rename()
```

The file access functions are:

```
fclose()     fflush()     fopen()     freopen()
```

The formatted input/output functions are:

```
fprintf()    fscanf()     printf()    scanf()
sprintf()    sscanf()
```

The character input/output functions are:

```
fgetc()      fgets()      fputc()     fputs()
getc()       getchar()    gets()      putc()
putchar()    puts()       ungetc()
```

The direct input/output functions are:

```
fread()      fwrite()
```

The file positioning functions are:

```
fseek()      ftell()      rewind()
```

`<stdlib.h>` This header declares two types and several functions of general utility, and defines four macros.

The macros are `ERANGE`, `HUGE_VAL`, and `RAND_MAX`.

The string conversion functions are:

```
atof()      atol()      strtod()    strtol()
atoi()      itoa()      ltoa()
```

The pseudo-random sequence generator functions are:

```
rand()      srand()
```

The memory management functions are:

```
calloc()    free()      malloc()    realloc()
```

The DOS communication functions are:

```
abort()     exit()      getenv()    system()
atexit()
```

The searching and sorting functions are: `bsearch()` `qsort()`

The integer arithmetic functions are:

```
abs()       labs()
```

`<string.h>` This header declares several functions for manipulating character arrays.

The copying functions are:

`memccpy()`	`memcpy()`	`memmove()`	`strcpy()`
`strncpy()`			

The concatenation functions are:

`strcat()`	`strncat()`

The comparison functions are:

`memcmp()`	`strcmp()`	`strncmp()`

The search functions are:

`memchr()`	`strcspn()`	`strrchr()`	`strstr()`
`strchr()`	`strpbrk()`	`strspn()`	`strtok()`

The miscellaneous functions are:

`memset()`	`strlen()`

abort

`void abort(void)`

abort prints the message

```
Abnormal program termination
```

to `stderr`, and then exits to DOS.

RETURNS: *abort* does not return to the caller. A return-code of 3 is returned to DOS.

EXAMPLE:

```
FILE *mustopen(char *name, char *mode) {
  FILE *fd;

  if((fd = fopen(name, mode)) == NULL) {
    printf("can't open %s", name);
    abort();
    }
  return fd;
  }
```

abs

`#include <stdlib.h>`

`int abs(int n)`

abs computes the absolute value of *n*..

RETURNS: *abs* returns the `int` absolute value of its integer argument. There are no error values.

SEE ALSO: `fabs(), labs()`

EXAMPLE:

```
int x;

if(x != abs(x))
  puts("negative");
```

access

`int access(char *path, int mode)`

The *access* function tests for the existence of the file specified by *path* , and whether it can be accessed in *mode* . The values and meanings of *mode* are:

Value	Meaning
0	Check for existence only
2	Write access
4	Read access
6	Read and write access

mode 0 and 4 produce the same result, since all DOS files have read access. *mode* 2 and 6 produce the same result for the same reason.

RETURNS: *access* returns 0 if the file can be accessed with the specified *mode* . *access* returns -1 if *path* doesn't exist or can't be accessed in the specified *mode.*

EXAMPLE:

```
char comspec[65];

getenv("COMSPEC", comspec, sizeof(comspec));
if(access(comspec, 0)) {
  puts("whoops, command.com is missing");
  abort();
  }
```

acos

`#include <math.h>`

`double acos(double x)`

acos computes the arc-cosine of *x* in the range 0 to π. *x* must be between -1.0 and 1.0.

RETURNS: *acos* returns the arc-cosine of its argument. *acos* returns 0.0 and sets *errno* to EDOM for *x* > 1.0 or *x* < -1.0.

EXAMPLE:
```
#include <math.h>
double x,y;

if(x < -1.0 || x > 1.0)
  printf("%g is invalid acos() argument\n", x);
else
  y = acos(x);
```

asin

`#include <math.h>`

`double asin(double x)`

asin computes arc-sin of *x* in the range $-\pi/2$ to $\pi/2$. *x* must be between 1.0 and -1.0.

RETURNS: *asin* returns the arc-sine of its argument. *asin* returns 0.0 and sets *errno* to **EDOM** for $x > 1.0$ or $x < -1.0$.

EXAMPLE:

```
#include <math.h>
double x,y;

if(x < -1.0 || x > 1.0)
   printf("%g is invalid asin() argument\n", x);
else
   y = asin(x);
```

assert

`#include <assert.h>`

`void assert(`*expr***`)`**

assert prints a diagnostic message and terminates the program if *expr* evaluates to 0 (`FALSE`). The message has the following format:

`Assertion (`*expr*`) failed: file `*name*`, line `*number*

name is the name of the source file containing the *assert* macro. *number* is the line number of the *assert* macro in *name*.

The `#line` preprocessor directive can alter both *name* and *number*.

No action is taken if *expr* evaluates to non-zero (`TRUE`).

If the macro name `NDEBUG` has been defined, the preprocessor removes all *assert* macros from the source file.

RETURNS: There is no return value.

EXAMPLE:

```
#include <assert.h>
#include <math.h>

double tasin(double x) {

  assert(x >= -1.0 && x <= 1.0);
  return asin(x);
  }
```

atan — atan2

```
#include <math.h>

double atan(double x)

double atan2(double x, double y);
```

atan computes the arc tangent of *x* in the range of -π/2 to π/2.

atan2 computes the arc tangent of *y* / *x* in the range of -π to π.

RETURNS: *atan* and *atan2* returns the arc tangent of their argument(s). If both arguments of *atan2* are zero, the function sets `errno` is `EDOM` and returns `0.0`.

EXAMPLE:

```
/*
** return arc tangent in degrees
*/

#include <math.h>

extern double PI;

double atan_deg(x)
double x;{
  return atan(x) * 180 / PI;
  }
```

atexit

`#include <stdlib.h>`

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

atexit registers the function pointed to by *func* , to be called without arguments at normal program termination.

Up to 32 functions may be registered.

RETURNS: *atexit* returns 0 if the function was registered, non-zero otherwise.

EXAMPLE:

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

/* send eof message to remote */

void eop(void) {
   fputc(EOF, stdaux);
   }


main(){
   if(atexit(eop)) {
      puts("atexit() error");
      abort();
      .
      .
      .
   }
```

atof

```
#include <math.h>

double atof(cp)
char cp[];
```

atof converts the `char` array at *cp* to a `double`. The first unrecognized character terminates the conversion. There is no test for overflow.

whitespace is either a tab or a space. A *digit* is an ASCII character '0' through '9'. *E* is either an ASCII 'E' or 'e'. [] delimit sequences that can occur zero or one time. {} delimit sequences that can occur zero or many times.

Valid character Sequences

{*whitespace* }[-]{*digit* }['.'][{*digit* }[*E* [-]{*digit* }]]

RETURNS: *atof* returns the floating-point representation of the string *cp* , or 0.0 if the string couldn't be converted. There are no error values.

SEE ALSO: `scanf()`

EXAMPLE:

```
/* floating-point scanner */

#include <math.h>

double scand(){
  char buf[80];

  gets(buf);  /* allow line editing */
  return atof(buf);
  }
```

atoi

```
int atoi(cp)
char cp[];
```

atoi converts the `char` array at *cp* to an `int`. The first unrecognized character terminates the conversion. There is no test for overflow.

whitespace is either a tab or a space. A *digit* is an ASCII character '0' through '9'. [] delimit sequences that can occur zero or one time. {} delimit sequences that can occur zero or many times.

Valid character Sequence

{*whitespace* } [-] {*digit* }

RETURNS: *atoi* returns the integer representation of the string *cp*, or 0 if the string couldn't be converted. There are no error values.

SEE ALSO: `atol(), scanf()`

NOTES: To guard against `int` overflow, or large `unsigned` values being stored as negative integers, use *atol* and check that the high-order word is the same as the high-order bit of the low-order word.

```
#include <math.h>

atoi(str)
char *str; {
  long val, atol();
  unsigned sign, extn;
  extern int errno;

  extn = (val = atol(str)) >> 16;
  sign = val & 0x8000;
  if((!sign && extn != 0)
     || (sign && extn != -1))
       rerrno(errno = ERANGE);
  return val;
  }
```

atol

```
#include <math.h>

long atol(cp)
char cp[];
```

atol converts the `char` array at *cp* to a `long`. The first unrecognized character terminates the conversion. There is no test for overflow.

whitespace is either a tab or a space. A *digit* is an ASCII character '0' through '9'. [] delimit sequences that can occur zero or one time. {} delimit sequences that can occur zero or many times.

Valid character Sequences

{*whitespace* } [-] {*digit* }

RETURNS: *atol* returns the integer representation of the string *cp* , or 0L if the string couldn't be converted. There are no error values.

SEE ALSO: `scanf()`

EXAMPLE: See ***atoi***

bsearch

```
#include <stdlib.h>

void *bsearch(const void *key,
        const void *base,
        size_t nmemb, size_t size,
        int (*cmp)(void*, void*) );
```

bsearch searches an array of *nmemb* objects, the initial member of which is pointed to by *base* , for a member that matches the object pointed to by *base* . The size of each object is specified by *size* .

The array must be in ascending sorted order according to a comparison function pointed to by *cmp* , which is called with two arguments that point to the objects being compared. *cmp* should return one of the following values:

Value	Meaning
<0	first argument is less than second argument
0	first argument is equal to second argument
>0	first argument is greater than second argument

RETURNS: *bsearch* returns a pointer to the matching member of the array, or a NULL pointer if no match is found.

EXAMPLE:

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

struct kwords { char *name; short value; }
  kwords[] = { ...};

#define SZKWD sizeof(struct kwords)
#define NKWDS sizeof(kwords)/SZKWD

int cmp(struct kwords *l, struct kwords *r) {
  return strcmp(l->name, r->name);
  }

#define kwlook(p) \
  bsearch((p), kwords, NKWDS, SZKWD, cmp)
```

calloc

`#include <stdlib.h>`

`char *calloc(unsigned num, unsigned size)`

calloc allocates a block of *num* * *size* bytes. Each byte in the block is set to 0x00.

See the **Memory Management** discussion of the memory allocation functions.

RETURNS: *calloc* returns a pointer to the allocated block, or `NULL` if it couldn't allocate the memory.

SEE ALSO: `free(), freeall(), malloc(), realloc()`

EXAMPLE:

```
/* set up float array */

float *farray;

if(!(farray = calloc(500, sizeof(float))))
  error("no room for farray");
```

clearerr

`#include <stdio.h>`

`void clearerr(FILE *stream);`

clearerr clears the end-of-file and error indicators for *stream* . These indicators are cleared only when the file is opened or by an explicit call to *clearerr* or *rewind* .

RETURNS: *clearerr* returns no value.

NOTE: *clearerr* is a macro.

EXAMPLE:

```
#include <stdio.h>

int output(FILE *fp, char c) {

  if(fputc(c, fp) == EOF && ferror(fp)) {
    fputs("write error\n", stderr);
    clearerr(fp);
    }
  }
```

ceil

```
#include <math.h>

double ceil(x)
double x;
```

ceil returns a `double` with the smallest integer value greater than or equal to `x`.

RETURNS: *ceil* returns the ceiling of *x*. There are no error values.

Note that `ceil(-1.5)` == -1.0, not -2.0.

SEE ALSO: `floor(), fmod()`

EXAMPLE:

```
/* round away from zero */

#include <math.h>

double rndfr0(x)
double x:{

  if(x < 0.0)
     return -ceil(fabs(x);
  return ceil x;
  }
```

chain

```
void chain(filename, commandTail)
char filename[], commandTail[];
```

chain functions like *exec* except that control is not returned to the calling program. *chain* is in EXEC.O on the distribution disk. It should be bound in first to save memory since it loads the called program immediately behind itself. For example:

```
    A>BIND EXEC BLIP -OBLIP
```

See the discussion in **System Interface** for more information on *chain.*

RETURNS: *chain* does not return.

SEE ALSO: `exec()`

EXAMPLE:

```
/* link to assembler
** assemble file myprog.a in current directory
** use large case flag
*/

chain("c:\\c88\\asm88.exe", "myprog b");
```

chdir

```
int chdir(pathname)
char pathname[];
```

chdir changes the current working directory to *pathname*. *pathname* must exist and be a directory.

RETURNS: *chdir* returns a 0 if successful, or -1 if it fails.

NOTE: *pathname* is of the form [*drive* :] [*path*]. Either, or both components may be specified.

REMEMBER — the backslash "\" is the escape character in C. If you are specifing *pathname* at COMPILE time, use two backslashes.

```
chdir("c:\\c88");
```

Since DOS accepts either a slash or a backslash as a path separator, the following is equivalent.

```
chdir("c:/c88");
```

If the *pathname* is entered at RUN time, only a single slash or backslash is needed.

EXAMPLE:

```
char path[128];

puts("Enter New Directory ->");
gets(path);
if(strlen(path))
   if(chdir(path) != 0)
      patherror(path);
```

chmod

```
#include <dos.h>

int chmod(filename, mode)
char filename[];
int mode;
```

chmod changes the attributes of *filename* as specified by *mode*. *filename* must exist.

mode is a value containing a zero or one or both of the constants CHDIR_READONLY and CHDIR_HIDDEN specified in *dos.h*. Both constants can be specified as CHDIR_READONLY+CHDIR_HIDDEN.

Value	Writable	Hidden
0	yes	no
CHDIR_READONLY	no	no
CHDIR_HIDDEN	yes	yes
CHDIR_READONLY+ CHDIR_HIDDEN	no	yes

RETURNS: *chdir* returns a 0 if successful, or -1 if *filename* could not be found.

EXAMPLE:

```
/* program to hide files
** run as hide fn fn ...
*/

#include <dos.h>

main(argc, argv)
char *argv[];{
  int i;

  for(i = 1; i < argc; i++)
    chdir(argv[i], CHDIR_HIDDEN);
  }
```

ci

`char ci();`

ci reads the next character from the keyboard. If one is not available, *ci* waits until one is entered. The character is not echoed to the display, and is not retained for future access via `getc()`....

Input to *ci* can be redirected.

There is no check for CTRL-C.

SEE ALSO: `csts(), getchar(), scr_ci()`

NOTES: *ci* returns a zero as the first character of an extended key sequence, and returns the extended key code on the next call.

To decode an extended key sequence, use `scr_ci()`. It maps the extended key sequences into `char` values between `0x80` and `0xFF`. See the CONFIG.C and PCIO.A files for the mapping.

EXAMPLE:

```
/* get password from kybd */

void getpass(word, len)
char word[];
int len;{
   char *cp = word;

   while(len-- && (*cp++ = ci()) != '\r')
      ;
   *cp = '\0';
   }
```

close

```
int close(handle)
int handle;
```

close writes any buffered data for the file associated with *handle* to that file, and closes the file.

RETURNS: *close* returns `-1` if `handle` is invalid, not open, or if an error occurred writing the buffered data to that file.

SEE ALSO: `closeall(), creat(), dup(), dup2(), open()`

EXAMPLE:

```
/* copy file to console */

main(argc, argv)
char *argv[];{
   int fd, sz;
   char buf[2048], *bp;

   if((fd = open(argv[1], 0)) == 0){
      printf("can't open %s", argv[1]);
      exit(1);
      }
   while(sz = read(fd, buf, sizeof(buf)))
      for(bp = buf; sz; sz--)
         co(*bp++);
   close(fd);
   }
```

closeall

`int closeall()`

closeall flushes all buffers and closes all open files.

RETURNS: *closeall* returns `-1` if an error occurred closing any file.

SEE ALSO: `close(), creat(), dup(), dup2(), open()`

EXAMPLE:

```
/* fatal error routine */

#include <stdio.h>

void fatal(msg, ret)
char *msg;{

   fputs(msg, stderr);
   closeall();
   exit(ret);
   }
```

CO

```
void co(ch)
char ch;
```

co writes the character *ch* on the screen at the current cursor position. The cursor is advanced to the next position on the screen. There is no automatic conversion of the newline character `\n` into the `\r\n` (carriage return, line feed) sequence needed by the screen driver.

co output can be redirected.

No test for CTRL-C is performed.

SEE ALSO: `putchar(), scr_co()`

EXAMPLE:

```
/* puts() equivalent */

void puts(string)
char *string;{
  char ch;

  while(ch = *string){
    if(ch == '\n')
      co('\r');
    co(*string++);
    }
  }
```

COS

```
#include <math.h>

double cos(x)
double x;
```

cos computes the cosine of its radian argument *x*. The meaningfulness of the result depends upon the magnitude of the argument.

RETURNS: *cos* returns the cosine of *x*. There are no error values.

SEE ALSO: `acos(), asin(), atan(), sin(), tan()`

EXAMPLE:

```
/* cos with degree argument */

#include <math.h>

extern double PI;

double dcos(x)
double x;{

  return cos((PI + PI) * x);
  }
```

creat

```
int creat(name);
char *name;
```

creat creates the file *name*. If *name* doesn't exist, a new file is created. If the file exists, its contents are deleted.

The file is opened in update mode so that after the file is written, a program can seek to its begining and read it without closing the file and reopening it.

RETURNS: *creat* returns an handle that is used to reference the file in future operations, or -1 if the file couldn't be opened.

SEE ALSO: `dup(), dup2(), open`

NOTE: *creat* can open the console (`"CON"`), the serial port (`"AUX"`), or the printer (`"PRN"`).

EXAMPLE:
```
int fd;

if((fd = creat("c:\\templ")) == -1)
   fatal("can't create templ", 5);
```

csts

`char csts();`

csts is similar to *ci* except that if no character has been typed in, it will return zero instead of waiting for a character from the keyboard.

The character is retained and will be returned by the next call to *ci*.

SEE ALSO: `getchar(), scr_ci(), scr_csts()`

NOTES: *csts* returns a zero as the first character of an extended key sequence, and returns the extended key code on the next call.

In order to decode an extended key sequence, use `scr_csts()`. It maps the extended key sequences into `char` values between `0x80` and `0xFF`. See the files CONFIG.C and PCIO.A for the mapping.

csts leaves the character in the input queue. Use *ci* or *getchar* to read it.

EXAMPLE:

```
/* empty the input queue
**
** note - doesn't work for extended key codes
**        use scr_csts()
*/

void emptyKbd(){
     while(csts())
        ci();
   }
```

dates

```
void dates(buf);
char buf[9];
```

dates formats the string *buf* with the current date as "`mm/dd/yy`".

If `mm` or `dd` are less than 10, they will be formated with a space (0x20) as their first character.

RETURNS: *dates* doesn't return a value.

SEE ALSO: `ctime()`

EXAMPLE:

```
/* current date */

char *currdate(){
  static char cdate[9];

  if(cdate[0] == 0)
    dates(cdate);
  return cdate;
  }
```

dup — dup2

```
int dup(file1)
int file1;

int dup2(file1, file2)
int file1, file2;
```

dup creates a second file handle for the open file *file1*. Either handle can be used to operate on the file.

dup2 forces *file2* to refer to the same file as *file1*. If *file2* refers to an open file, that file is closed.

RETURNS: *dup* returns the new file handle, or -1 if an error occurs.

dup2 returns 0, or -1 if an error occurs.

SEE ALSO: `close(), creat(), open()`

EXAMPLE:

```
/* redirected i/o */

#include <stdio.h>

int printer, fno;
char pname[32];

sprintf(pname, "c:\\spool\\F%d.pfl", fno++);
if((printer = creat(pname)) == -1)
  fatal("can't allocate printer", 1);
if(dup2(printer, stdout))
  fatal("can't redirect to printer", 2);
exec(command, parms);
close(printer);
```

exec

```
char exec(filespec, commandTail)
char filespec[], commandTail[];
```

exec loads and executes an arbitrary program —*filespec* is the complete pathname of the program (including the .EXE or .COM extension). *commandTail* contains the arguments to the program. *exec* is in the EXEC.O file provided on the distribution disks.

RETURNS: *exec* returns the completion code from the program, or -1 if an error occurred loading the program. Completion codes are set for programs running under DOS 2.0 or greater. If a program exits with

```
exit(n);
```

the system ERRORLEVEL will be set to n. A program that returns from the `main` function other than by `exit()` sets ERRORLEVEL to zero. ERRORLEVEL can be tested with the DOS batch file IF command. See the section under 'BATCH' in the DOS manual for details on the IF command.

SEE ALSO: `chain(), exit()`

EXAMPLE:

```
/*
** invoke command.com to process batch file
**
** use /c switch
*/

dobatch(fname)
char fname[];{
     char shell[64], parms[128];

     if(getenv("COMSPEC", shell) == 0)
        return 256;  /* getenv() error */
     strcpy(parms, "/c");
     strcat(parms, fname);
     return exec(shell, parms);
     }
```

exit

```
void exit(code)
char code;
```

exit terminates the calling process and sets the completion *code*. `main()` can also *exit* with a completion code of zero by "falling" through the end of the function.

Only the low-order byte of *code* is used.

exit does <u>not</u> close open files.

RETURNS: *exit* does not return.

SEE ALSO: `exec(), chain()`

NOTE: Completion codes are set for programs running under DOS 2.0 or greater. If a program exits with

```
     exit(n);
```

the system ERRORLEVEL is set to *n* . A program that returns from the `main` function other than by *exit* sets ERRORLEVEL to zero. ERRORLEVEL can be tested with the DOS batch file IF command. See the section under 'BATCH' in the DOS manual for details on the IF command.

EXAMPLE:

```
/* fatal error handler */

#include <stdio.h>

fatal(msg, level)
char *msg, level;{

   fputs(msg, stderr);
   exit(level);
   }
```

exp, exp10

```
#include <math.h>

double exp(x)
double x;

double exp10(x)
double x;
```

exp returns the exponential function of *x*; *exp10* returns the base 10 exponent.

RETURNS: *exp* returns e^x.

exp10 returns 10^x.

Both return a very large value when the result would overflow; *errno* is set to **ERANGE**.

SEE ALSO: `log(), log10()`

EXAMPLE:

```
#include <math.h>

double x;

x = exp(x);
if(errno = ERANGE)
  puts("This isn't a CRAY II\n");
else
  printf("%f", x);
```

fabs

```
#include <math.h>

double fabs(x)
double x;
```

fabs computes the absolute value of *x* .

RETURNS: *fabs* returns the absolute value of its `double` argument. There are no error codes set.

SEE ALSO: `abs(), labs()`

EXAMPLE:

```
#include <math.h>

double x;

if(x != fabs(x))
  puts("negative");
```

fclose

```
#include <stdio.h>

int fclose(fp);
FILE *fp;
```

fclose writes any buffered data for the file `fp` to that file and closes the file

RETURNS: *fclose* returns 0 , or -1 if `fp` is not open or if an error occurred writing the buffered data to that file.

SEE ALSO: `close(), fflush(), fopen(), freopen()`

EXAMPLE:

```
/* file copy */
#include <stdio.h>

static fatal(msg, level)
char *msg, level;{
     fputs(msg, stderr);
     exit(level);
     }

main(argc, argv)
char *argv[];{
   char buf[1024];
   FILE *ip, *op;

   if(argc != 3){
      fatal("Usage: copy name name\n", 1);
   if(*argv[1] == '-')
      ip = stdin;
   else if((ip = fopen(argv[1], "r")) == NULL)
      fatal("can't open input file", 2);
   if(*argv[2] == '-')
      op = stdout;
   else if((ip = fopen(argv[2], "w")) == NULL)
      fatal("can't open output file", 3);
   while(fgets(buf, sizeof(buf), ip))
      fputs(buf, op);
   fclose(op); /* write buffered data out */
   }
```

feof

```
#include <stdio.h>

int feof(FILE *stream);
```

feof tests the end-of-file indicator for *stream* .

RETURNS: *feof* returns a non-zero value it the end-of-file indicator is set for *stream* .

NOTE: *feof* is a macro.

EXAMPLE:

```
#include <stdio.h>

/* line oriented file copy */

main(){
    char buffer[1024];

    while(1) {
        gets(buffer);
        if(feof(stdin))
            break;
        puts(buffer);
        }
    }
```

ferror

```
#include <stdio.h>

int ferror(FILE *stream);
```

ferror tests the error indicator for *stream* .

RETURNS: *ferror* returns a non-zero value it the error indicator is set for *stream* .

NOTE: *ferror* is a macro.

EXAMPLE:

```
#include <stdio.h>
int output(FILE *fp, char c) {

   if(fputc(c, fp) == EOF && ferror(fp)) {
      fputs("write error\n", stderr);
      clearerr(stream);
      }
   }
```

fflush

```
#include <stdio.h>

int  fflush(fp);
FILE *fp;
```

fflush writes any buffered data for the file `fp` to that file. The file remains open.

RETURNS: *fflush* returns 0, or -1 if `fp` is not open or if an error occurred writing the buffered data to the file.

SEE ALSO: `fclose()`

NOTE: Only disk files are buffered, so *fflush* does nothing on non-disk files.

EXAMPLE:

```
/*
** write record to file and
** release to network
*/

#include <stdio.h>
#include <dos.h>

nwrite(buf, size, recno, fp)
char buf[];
int size, recno;
FILE *fp;{
  int fno;

  fseek(fp,(long)size * (long)recno, 0);
  locking(fno = fileno(fp), LOCK, size);
  fwrite(buf, size, 1, fp);
  fflush(fp);
  locking(fno, UNLCK, size);
  }
```

fgetc

```
#include <stdio.h>

int fgetc(fp)
FILE *fp;
```

fgetc returns the next character from the stream *fp*. *fp* must be open.

RETURNS: *fgetc* returns the next character from the stream, or `EOF` on error or end-of-file.

SEE ALSO: `getc(), scanf(), fread()`

EXAMPLE:

```
/*
** fgets
*/

#include <stdio.h>

char * fgets(buf, n, fp)
char buf[];
int n;
FILE *fp;{
  int i, ch;

  n--;
  for(i = 0; i < n && ((ch = fgetc(fp)) != EOF)
    && (ch != '\r' || ch != '\n'); i++)
          buf[i] = ch;
  if(i != n)
    buf[i++] = '\n';
  buf[i] = '\0';
  while(ch == '\r' || ch == '\n')
    ch = fgetc(fp);
  return buf;
  }
```

fgets

```
#include <stdio.h>

char *fgets(buf, len, fp)
char buf[];
int len;
FILE *fp;
```

fgets reads the next line, but not more than *len* - 1 characters from the file *fp* into *buf*. The last character read into *buf* is followed by a `'\0'`.

RETURNS: *fgets* returns *buf*, or `NULL` on end of file or an error.

SEE ALSO: `fscanf(), fread()`

NOTE: *fgets* returns the CR character.

EXAMPLE:

```
/*
** copy file to console
*/

#include <stdio.h>

void f_to_con(fp)
FILE *fp;{
   char buf[81];

   while(fgets(buf, 81, fp))
      puts(buf);
   }
```

filelength

```
long filelength(handle)
int handle;
```

filelength accesses the size of the open file associated with *handle*.

RETURNS: *filelength* returns the actual length of the file in bytes, or -1L on error.

SEE ALSO: `fileno()`

EXAMPLE:

```
/*
** reprocess additions
*/

process(fh)
int fh;{
   long restart, filelength();

   restart = filelength(fh);
   doAdditions(fh);
   lseek(fh, restart, 0);
   reprocess(fh);
   }
```

fileno

```
#include <stdio.h>

int fileno(fp)
FILE *fp;
```

fileno returns the handle associated with the file *fp*. If more than one handle is linked to the file *fileno* returns the handle assigned when the file was opened.

RETURNS: *fileno* returns the file handle. The value is undefined if there is no file associated with *fp*.

SEE ALSO: `filelength(), fopen(), freopen()`

EXAMPLE:

```
/*
** Find length of a stream
/*

#include <stdio.h>

long flen(fp)
FILE *fp;{
  long filelength();

  return filelength(fileno(fp));
  }
```

floor

```
#include <math.h>

double floor(x)
double x;

     dval = floor(x);
```

floor returns a `double` with the largest integer value less than or equal to x.

RETURNS: *floor* returns the floor of *x*. There are no error values.

Note that `floor(-1.5)` = -2.0, not -1.0.

SEE ALSO: `ceil()`

EXAMPLE:

```
/* round towards zero. cf. ceil() */

#include <math.h>

double rndto0(x)
double x:{

  if(x < 0.0)
    return -floor(fabs(x));
  return floor(x);
  }
```

fopen

```
#include <stdio.h>

FILE *fopen(name, method)
char *name, *method;
```

fopen opens the file *name* .

method is a string having one of the following values:

Method	Meaning
"r	open for reading (file must exist).
"w"	open for writing (same as *creat*).
"a"	open for append — open for writing at end of file, or create for writing.

RETURNS: *fopen* returns a `FILE*` that identifies the file in future file operations, or returns `NULL` if the file couldn't be opened.

NOTES: Even though *fopen* can open the console (`"CON"`); the serial port (`"AUX"`), or the printer (`"PRN"`), you save file handles by using the standard files (`stdin`, `stdout`, ...).

EXAMPLE:

```
/*
** initialize (possibly) empty file
*/

#include <stdio.h>

FILE *finit(name){
char name[];{
   long ftell();
   FILE *fp;

   if((fp = fopen(name, "a")) == 0)
      error("can't open %s", name);
   if(ftell(fp) == 0L)
      fileInit(fp);
   return fp;
   }
```

FP_OFF, FP_SEG
(Large Case Option)

```
#include <dos.h>

unsigned FP_OFF(ptr)
char *ptr;

unsigned FP_SEG(ptr)
char *ptr;
```

FP_OFF and *FP_SEG* are macros that decompose an 8088 physical address (SEG:OFF) into its constituent parts.

FP_OFF and *FP_SEG* work only with Large Case programs.

RETURNS: *FP_OFF* returns the offset component of the 8088 physical address; *FP_SEG* returns the segment component.

EXAMPLE:

```
/*
** Large (>64K) array addressing
**
** base address assumed to be SEG:0
*/

#include <dos.h>

char *larray(ptr, off)
char *ptr;              /* base adr of array */
long off;{              /* offset into array */
  long addr;

  addr  = (long)(FP_SEG(ptr)) << 16;
  addr |= (off & 0xFFFF0L) << 12;
  addr |= off & 0xFL;
  return addr;
  }
```

fprintf

```
#include <stdio.h>
int fprintf( fp, fcs, [, arg ] ...)
FILE *fp;
char fcs[];
```

fprintf formats the data in *fcs* and *[arg]* to the file *fp*, which must be open.

The format control string, *fcs*, can contain both ordinary characters which are copied unchanged to the output file, and conversion control strings which describe how each *arg* is to be formatted. *fcs* is specified in *printf*.

RETURNS: *fprintf* returns -1 on error.

NOTE: The maximum length of *fprintf* output is 256 bytes. If you need more use *sprintf* followed by fputs().

EXAMPLE:

```
/*
** write <name, index, val> in text format
*/

#include <stdio.h>

Twrite(fp, n, i, v)
FILE *fp;
char n[];
double v;{
   return fprintf(fp,"\"%s\",%d,%f\n", n, i, v);
   }
```

fputc

```
#include <stdio.h>

int fputc(ch, fp)
char ch;
FILE *fp;
```

fputc writes *ch* to the file *fp* . *fp* must be open.

RETURNS: *fputc* returns *ch* , or -1 on error.

SEE ALSO: `printf(), putc(), fwrite()`

EXAMPLE:

```
/*
** u**x puts
*/

#include <stdio.h>

static int DOSputc(ch, fp)
char ch;
FILE *fp;{
   if(ch == '\n')
      fputc('\r', fp);
   return fputc(ch, fp);
   }

uputs(str)
char *str;{
   char ch;

   while(ch = *str++)
      if(DOSputc(ch, stdout) == EOF)
         return EOF;
   return DOSputc('\n', stdout);
   }
```

fputs

```
#include <stdio.h>

int fputs(buf, fp);
char buf[];
FILE *fp;
```

fputs copies the string *buf* to the file *fp*.

RETURNS: *fputs* returns a -1 on error.

SEE ALSO: `fprintf(), fwrite()`

NOTE: *fputs* converts linefeed ('\n') to carriage return - linefeed ('\r\n'). Output will stop if CTRL-S is entered, and resume when any other key is pressed. Each output will check for a CTRL-C entry, and terminate the program if one is encountered.

EXAMPLE:

```
/* file copy */
#include <stdio.h>

main(argc, argv)
char *argv[];{
  char buf[1024];
  FILE *ip, *op;

  if(*argv[1] == '-')
    ip = stdin;
  else if((ip = fopen(argv[1], "r")) == NULL)
    exit(1);
  if(*argv[2] == '-')
    op = stdout;
  else if((op = fopen(argv[2], "w")) == NULL)
    exit(2);
  while(fgets(buf, sizeof(buf), ip))
    fputs(buf, op);
  fclose(op); /* write buffered data out */
  }
```

frand

`double frand();`

frand computes the next pseudo-random number.

RETURNS: *frand* returns the next pseudo-random number in the range from 0.0 to 1.0. There are no error values.

SEE ALSO: `rand(), srand()`

EXAMPLE:

```
/*
** draw element from the set [min, max]
*/

draw(min, max)
int min, max;{
   double ceil(), frand();

   return min + ceil((max - min) * frand());
   }
```

fread

```
#include <stdio.h>

unsigned fread(buf, size, nitems, fp)
char buf[];
unsigned size, nitems;
FILE *fp;
```

fread reads into *buf*, *nitems* of data of size *size*, from the file *fp* .

RETURNS: *fread* returns the number of items actually read (which may be less than *nitems* if end-of-file is encountered), or 0 if an error occurred.

SEE ALSO: `fgetc(), fgets(), scanf()`

free, freeall

```
void free(op)
char *op;

void freeall(stackSize)
unsigned stackSize;
```

free marks the block at *op* as unallocated.

freeall reserves *stack* bytes for the stack expansion area, and initializes the memory allocation area.

See the **Memory Management** section for a discussion of the memory allocation area.

RETURNS: Neither *free* or *freall* return a value. No error codes are set.

NOTE: *freeall* releases any storage allocated by *malloc*, ... You can reset the memory allocation area by calling *freeall*. Otherwise, call *freeall* prior to any calls to *malloc*, ...

SEE ALSO: `calloc(), malloc(), realloc()`

EXAMPLE:

```
/*
** reserve 20k for stack expansion
*/

freeall(20 * 1024);

/*
** process a line
*/

if((cp = malloc(MAXLINE)) && process(cp))
     free(cp);
```

freopen

```
#include <stdio>

FILE *freopen(filename, method, fp)
char *filename, *method;
FILE *fp;
```

freopen closes the file associated with *fp* and redirects to the file *name* . It is normally used to redirect `stdin`, `stdout`, `stderr`, `stdaux`, and `stdprn`.

method is a `char` string having one of the following values:

Method	Meaning
"r	open for reading (file must exist).
"w"	open for writing (same as *creat*).
"a"	open for append — open for writing at end of file, or create for writing.

RETURNS: *freopen* returns an `FILE*` that identifies the file in future file operations, or returns `NULL` if the file can't be redirected.

EXAMPLE:

```
/*
** redirected i/o
*/

#include <stdio.h>

FILE *prn, *freopen();
char pname[32];
int fno;

sprintf(pname, "c:\\spool\\F%d.pfl", fno++);
if((prn = freopen(pname, "w", stdout)) == NULL)
  fatal("can't redirect to printer", 2);
exec(command, parms);
fclose(prnt);
```

frexp

```
#include <math.h>


double frexp(value, eptr)
double value;
int *eptr;
```

frexp disassembles *value* into a fraction (< 1.0), and its base 2 exponent.

RETURNS: *frexp* returns the fractional part of *value* as a double, and the base 2 exponent of *value* as an integer at **eptr*.

The value 0.0 returns both 0.0 as the fraction and 0 as the exponent.

There are no error codes set.

SEE ALSO: `ldexp(), modf()`

EXAMPLE:

```
/*
** multiply by power of 2
**

#include <math.h>

double mpyPow2(num, pwr)
double num;
int pwr;{
   int exp;

   num = frexp(num, &exp);
   return ldexp(num, exp + pwr);
   }
```

fscanf

```
#include <stdio.h>

int fscanf(fp, fcs [, ptr ] ...)
FILE *fp;
char fcs[];
```

fscanf reads from the file *fp*, assembles data under the specification of *fcs*, and stores the data at **ptr*.

The format control string, *fcs*, is described in *scanf.*

RETURNS: *fscanf* returns the number of fields scanned and assigned. A return of zero means no fields were converted.

fscanf returns EOF for error or end-of-file.

SEE ALSO: `scanf(), sscanf()`

EXAMPLE:

```
/*
** read next numeric field from fp
**
** if there is any non-numeric data in the
** stream, discard it
*/

#include <stdio.h>

FILE *fp;

int n;

while(fscanf(fp, "%d", &n) == 0)
   fgetc(fp);
```

fseek

```
#include <stdio.h>

long fseek(fp, offset, mode)
FILE *fp;
long offset;
int   mode;
```

fseek sets the location of the next input or output operation on the file *fp* as follows:

mode	Location
0	*offset* bytes from the begining of the file
1	*offset* bytes from the current location
2	*offset* bytes from the end of the file

offset may be either positive or negative.

If the resulting location is before the beginning of the file, it is set to the beginning; if it is after the end of the file, it is set to the end.

RETURNS: *fseek* returns the current location, or -1L if there was an error.

SEE ALSO: `ftell(), lseek(), rewind()`

EXAMPLE:

```
/*
** rewind to begining of file
*/

#include <stdio.h>

long rewind(fp)
FILE *fp;{

  return fseek(fp, 0L, 0);
  }
```

ftell

```
#include <stdio.h>

long ftell(fp)
FILE *fp;
```

ftell gets the current location of the file *fp* as the relative byte offset from the beginning of the file.

RETURNS: *ftell* returns the current location, or -1L if there was an error.

SEE ALSO: `fseek(), lseek(), rewind()`

EXAMPLE:

```
/*
** process all records in file
*/

#include <stdio.h>

process(fp, fun, siz)
FILE *fp;
int (*fun)();
int siz;{
   long eof, filelength(), fseek(), ftell();
   char *buf = malloc(siz);

   eof = filelength(fileno(fp));
   fseek(fp, 0L, 0);
   while(ftell(fp) < eof){
      fread(buf, 1, siz, fp);
      (*fun)(buf);
      }
   free(buf);
   }
```

fwrite

```
#include <stdio.h>

int fwrite(buf, size, nitems, fp);
char buf;
unsigned size, nitems;
FILE fp;
```

fwrite appends from *buf*, at most *nitems* of data of length *size*, to the file *fp* .

RETURNS: *fwrite* returns the number of items actually written, which may be less than *nitems* if an error occured.

SEE ALSO: fputc(), fputs(), printf()

EXAMPLE:

```
/*
** write Large Case huge array to stream
** return: 0  == OK, otherwise error
*/

hwrite(at, size, fp)
char *at;           /* array */
long size;          /* size in bytes */
FILE *fp;           /* stream */
  {int rc;          /* return code */
  char *larray();   /* see FP_OFF */

  while(size > 0xFFFF){
    if(fwrite(at, 1, 0xFFFF, fp) != 0xFFFF)
      return 1;
    at = larray(at, 0xFFFFL);
    size -= 0xFFFF;
    }
  if(size &&
    fwrite(at,1,(int)size,fp) != (int)size))
      return 1;
  return 0;
  }
```

getc, getchar

```
#include <stdio.h>

int getc(fp)
FILE *fp;

int getchar();
```

getc reads the next character from the file *fp*. *fp* must be open.

getchar reads the next character from *stdin*.

RETURNS: *getc* and *getchar* return the next character, or EOF on error or end-of-file. *getchar* returns EOF when a CTRL-Z character is read.

SEE ALSO: `scanf(), fread()`

NOTE: *getc* and *getchar* are functions rather than a macros.

getchar can hangup reading redirected input under DOS 2.X and higher. Use `getc(stdin)` if the input could be redirected.

EXAMPLE:

```
/*
** read integer from keyboard
**
** leave terminating char
*/

#include <stdio.h>

int geti(){
   char digits[128], *dp = digits;

   while(isdigit(*dp++ = getc(stdin)))
      ;
   ungetc(*(--dp), stdin);
   *dp = '\0';
   return atoi(digits);
   }
```

getdir

```
char *getdir(drive, pathBuffer)
char drive, pathBuffer[128];
```

getdir writes the full pathname of the current directory into *pathBuffer*.

drive is the drive number: 0 = default drive, 1 = A:, ...

pathBuffer should be a 128 character array.

RETURNS: *getdir* returns the address of *pathBuffer*, or -1 on error.

EXAMPLE:

```
/*
** display prompt as [hh:mm]path>
*/

prompt(){
   char pbuf[137];

   strcpy(pbuf, "\n[");
   times(&pbuf[2]);
   pbuf[7] = ']';
   getdir(0, &pbuf[8]);
   strcat(pbuf, ">");
   puts(pbuf);
   }
```

getenv

```
char *getenv(key, buffer)
char *key, buffer[80];
```

getenv searches the DOS environment for an entry of the form

key =value

and copies *value* into *buffer*. *value* is a string (terminates with '\0').

RETURNS: *getenv* returns the address of *buffer*, or `NULL` if *key* was not found.

NOTE: *key* is terminated by the '=' character, so

```
PATH=C:\
```

and

```
PATH =C:\
```

are different environment entries.

SEE ALSO: `putenv()`

EXAMPLE:

```
/*
** get path of COMMAND.COM
*/

#define COMSPEC(buf) getenv("COMSPEC",buf)
```

gets

```
char *gets(buf)
char buf[];
```

gets reads a line-edited string from the console (*stdin*) into *buf*. During input, <ESC> means backup and start over, <BACKSPACE> means delete the previous character and <RETURN> means end of string. <RETURN> is replaced in *buf* by a '\0'.

RETURNS: *gets* returns the address of *buf*, or NULL on end of file or error.

SEE ALSO: fscanf(), fread()

NOTE: *gets* doesn't return the CR character.

EXAMPLE:

```
/*
** copy a file from stdin to stdout
*/

cat(){
  buf[1024];

  while(gets(buf))
    puts(buf);
  }
```

getw

```
#include <stdio.h>

int getw(fp)
FILE *fp;
```

getw returns the next `int` from the file *fp*. *fp* must be open.

RETURNS: *getw* returns the next integer value, or `EOF` if an error or end of file was sensed.

NOTE: There is no way to distinguish the integer value -1 from EOF.

SEE ALSO: `scanf(), read()`

EXAMPLE:

```
/*
** Sum numbers in file
*/

#include <stdio.h>

long sum(fp)
FILE *fp;{
   long value = 0;
   int word;

   while((word = getw(fp)) != -1)
      value += word;
   return value;
   }
```

index

```
char *index(src, ch)
char src[], ch;
```

index finds the first occurence of *ch* in *src.*

index works on a null-terminated string. There is no test for overflow.

RETURNS: *index* returns a pointer to the first occurence of *ch* in *src,* or 0 if *ch* wasn't found.

EXAMPLE:

```
/*
** dispatch on key pressed
**
** use index rather than switch statement
*/

static char keys[] = "\003 ... ";

extern int ctl_c(), ... ;

static int (*fun)()[] = {ctl_c, ... };

dispatch(ch)
char ch[];{
     char *kp;
     extern char scr_attr;

     if(kp = index(keys, ch[0]))
          return (*fun)()[kp - keys];
     scr_aputs(ch, scr_attr);
     return 0;
     }
```

isalnum

```
int isalnum(c);
char c;
```

isalnum determines if `c` is a letter or a digit (A-Z, a-z, 0-9).

RETURNS: *isalnum* returns `TRUE` (non-zero) if `c` is a letter or a digit, `FALSE` (zero) otherwise.

NOTE: *isalnum* is a function rather than the usual macro implementation.

EXAMPLE:

```
/*
** is (c) alpha or numeric?
*/

isalnum(c)
char c;{
     return (c >= 'A' && c <= 'Z') ||
            (c >= 'a' && c <= 'z') ||
            (c >= '0' && c <= '9');
     }
```

isalpha

```
int isalpha(c);
char c;
```

isalpha determines if *c* is a letter (A-Z, a-z).

RETURNS: *isalpha* returns `TRUE` (non-zero) if *c* is a letter, `FALSE` (zero) otherwise.

NOTE: *isalpha* is a function rather than the usual macro implementation.

EXAMPLE:

```
/*
** is (c) alpha?
*/

isalpha(c)
char c;{
     return (c >= 'A' && c <= 'Z') ||
            (c >= 'a' && c <= 'z');
     }
```

isascii

```
int isascii(c);
char c;
```

isascii if *c* is an ASCII character (`0x00-0x7F`).

RETURNS: *isascii* returns `TRUE` (non-zero) if *c* is less than `0x80`, `FALSE` (zero) otherwise.

NOTE: *isascii* is a function rather than the usual macro implementation.

EXAMPLE:

```
/*
** is (c) an ascii char?
*/

isascii(c)
char c;{
     return (c & 0x80) == 0;
     }
```

isatty

```
int isatty(handle)
int handle;
```

isatty determines if the file *handle* refers to a character device — console, printer, or serial port.

RETURNS: *isatty* returns TRUE (non-zero) if the handle refers to a character device, FALSE (zero) otherwise.

EXAMPLE:

```
/*
** is (handle) a character device?
*/

#define ISDEV   0x0080
#define CHARDEV 0x8000

isatty(handle)
unsigned handle;{
     extern unsigned _rax, _rbx, _rdx;

     _rax = 0x4400; /* get device info */
     _rbx = handle;
     _doint(0x21);
     return (_rdx & ISDEV) && (_rdx & CHARDEV);
     }
```

iscntrl

```
int iscntrl(c)
char c;
```

iscntrl determines if *c* is a control character (`0x00-0x1F`, `0x7F`).

RETURNS: *iscntrl* returns `TRUE` (non-zero) if *c* is `0x7F` or less than `0x20` (space), `FALSE` (zero) otherwise.

NOTE: *iscntrl* is a function rather than the usual macro implementation.

EXAMPLE:

```
/*
** is (c) a control char?
*/

iscntrl(c)
char c;{
     return (c == 0x7F) ||
            (c <  0x20);
     }
```

isdigit

```
int isdigit(c);
char c;
```

isdigit determines if *c* is a digit (0-9).

RETURNS: *isdigit* returns `TRUE` (non-zero) if *c* is a digit, `FALSE` (zero) otherwise.

NOTE: *isdigit* is a function rather than the usual macro implementation.

EXAMPLE:

```
/*
** is (c) numeric?
*/

isdigit(c)
char c;{
     return (c >= '0' && c <= '9');
     }
```

islower

```
int islower(c);
char c;
```

islower determines if *c* is a lower-case letter (a-z).

RETURNS: *islower* returns `TRUE` (non-zero) if *c* is a lower-case letter, `FALSE` (zero) otherwise.

NOTE: *islower* is a function rather than the usual macro implementation.

EXAMPLE:

```
/*
** is (c) lower-case?
*/

islower(c)
char c;{
     return (c >= 'a' && c <= 'z');
     }
```

isprint

```
int isprint(c);
char c;
```

isprint determines if *c* is a printable character (`0x20-0x7E`).

RETURNS: *isprint* returns `TRUE` (non-zero) if *c* is a printable character, `0x20` (space) through `0x7E` ('~'), `FALSE` (zero) otherwise.

NOTE: *isprint* is a function rather than the usual macro implementation.

EXAMPLE:

```
/*
** is (c) printable?
*/

#define SPACE 0x20

isprint(c)
char c;{
     return (c >= SPACE && c <= '~');
     }
```

ispunct

```
int ispunct(c);
char c;
```

ispunct determines if *c* is neither a control nor an alphanumeric character.

RETURNS: *ispunct* returns `TRUE` (non-zero) if

```
!(isalnum(c) || iscntrl(c))
```

`FALSE` (zero) otherwise.

NOTE: *ispunct* is a function rather than the usual macro implementation.

isspace

```
int isspace(c);
char c;
```

isspace determines if *c* is a whitespace character (`0x09-0x0D`, `0x20`).

RETURNS: *isspace* returns `TRUE` (non-zero) if *c* is a `0x20` (space), `'\t'` (tab), `'\r'` (carriage return), `'\n'` (linefeed), or `'\f'` (formfeed), `FALSE` (zero) otherwise.

NOTE: *isspace* is a function rather than the usual macro implementation.

EXAMPLE:

```
/*
** is (c) white-space?
*/

isspace(c)
char c;{
     static char wspace[] = "\t\r\n\f\040";

     return index(wspace, c) != 0;
     }
```

isupper

```
int isupper(c);
char c;
```

isupper determines if *c* is an upper-case letter (A-Z).

RETURNS: *isupper* returns `TRUE` (non-zero) if *c* is an upper-case letter, `FALSE` (zero) otherwise.

NOTE: *isupper* is a function rather than the usual macro implementation.

EXAMPLE:

```
/*
** is (c) upper-case?
*/

isupper(c)
char c;{
     return (c >= 'A' && c <= 'Z');
     }
```

isxdigit

`#include <ctype.h>`

`int isxdigit(int c);`

isxdigit tests whether *c* is a valid hexadecimal digit [0-9,a-f,A-F].

RETURNS: *isxdigit* returns 1 if *c* is a valid hexadecimal digit, 0 otherwise.

EXAMPLE:

```
#include <ctype.h>

long gethex(char **p) {
    long val = 0;
    char c;

    while(isxdigit(**p)) {
        val <<= 4;
        if(isdigit(c = *(*p++)))
            val |= c - '0';
        else
            val |= toupper(c) - ('A' - 10);
        }
    return val;
    }
```

itoa

`#include <stdlib.h>`

`char *itoa(int val, char str[], int rad);`

itoa converts *val* into a null terminated string at *str* . *rad* specifies the base of *val* ; it must be in the range 2 — 36.

If *rad* is 10 and *val* is negative, the first character of *str* will be the minus sign, '-'.

RETURNS: *itoa* returns a pointer to *str*.

EXAMPLE:

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

/* put a decimal number to stdout */

putn(int dig) {
   char buffer[256];

   return fputs(itoa(dig, buffer, 10), stdout);
   }
```

labs

```
long labs(n)
long n;
```

labs computes the absolute value of *n*.

RETURNS: *labs* returns a `long` absolute value. There are no error codes set.

SEE ALSO: `abs(), fabs()`

EXAMPLE:

```
/*
** long absolute value
*/

long labs(val)
long val;{

   return val < 0 ? - val : val;
   }
```

ldexp

```
#include <math.h>

double ldexp(value, exp)
double value;
int exp;
```

ldexp computes *value* * 2^{exp}.

RETURNS: *ldexp* builds the floating-point representation. There is no test for overflow.

SEE ALSO: `frexp(), modf()`

EXAMPLE: See ***frexp***

locking

```
#include <dos.h>

int locking(handle, mode, count)
int handle, mode, count;
```

locking locks or unlocks *count* bytes of the file identified by *handle* starting at its current position. Locked areas of a file cannot be read or written by other processes.

mode specifies the action to be performed. The constants defined in *<dos.h>* are:

Constant	Action
`LOCK`	Lock the specified bytes. If the bytes cannot be locked, return an error.
`UNLCK`	Unlock the specified bytes. The bytes must be locked .

More than one area of a file can be locked, but the areas must not overlap.

Only one area can be unlocked per call. If two contiguous areas of the file are locked, each area must be unlocked separately.

All locks should be removed before closing the file or exiting the program.

RETURNS: *locking* returns 0 if successful, -1L on error.

SEE ALSO: `open()`

NOTE: *locking* works only with DOS 3.0 and later.

EXAMPLE: See *fflush*

log, log10

```
#include <math.h>

double log(x)
double x;

double log10(x)
double x;
```

log computes the natural logarithm of *x*; *log10* computes the base 10 logarithm.

RETURNS: *log* and *log10* return the indicated logarithms. They both return 0.0 when *x* is zero or negative and *errno* is set to EDOM.

SEE ALSO: `exp(), pow()`

EXAMPLE:

```
/*
** n-th root
*/

#include <math.h>

double root(n, x)
int n;
double x;{

   return exp( log(x) / (double)n );
   }
```

longjmp

```
#include <setjmp.h>

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

jmp_buf is defined in *<setjmp.h>*. It creates an environment used by *setjmp* for future use by *longjmp*. *longjmp* restores the environment from *env* and returns *val*. *val* cannot be zero.

RETURNS: *longjmp* does not return.

NOTE: *env* can be specified as zero for compatibility with previous releases. There can be only one "zero" *env* active at a time.

If the environment stored in *env* points into an overlay area, then the overlay that called *setjmp* must be resident when *longjmp* is called—if another overlay is resident, then strange things will happen. It is best to call *setjmp* from the root.

lseek

```
long lseek(handle, offset, mode)
int handle, mode;
long offset;
```

lseek sets the location of the next input or output operation on the file *handle* as follows:

mode	Location
0	*offset* bytes from the beginning of the file
1	*offset* bytes from the current location
2	*offset* bytes from the end of the file

offset may be either positive or negative.

If the resulting location is before the beginning of the file, it is set to the beginning; if it is after the end of the file, it is set to the end.

RETURNS: *lseek* returns the current location, or -1L if there was an error.

SEE ALSO: `fseek(), ftell()`

EXAMPLE:

```
/*
** ansi fseek()
**
** returns 0 == OK
*/

#include <stdio.h>

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

  ret = lseek(fileno(stream), offset, ptrname);
  return ret != offset;
  }
```

ltoa

```
#include <stdlib.h>

char *ltoa(long val, char str[], int rad);
```

ltoa converts *val* into a null terminated string at *str* . *rad* specifies the base of *val* ; it must be in the range 2 — 36.

If *rad* is 10 and *val* is negative, the first character of *str* will be the minus sign, '-'.

RETURNS: *ltoa* returns a pointer to *str*.

EXAMPLE:

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

/* put a decimal number to stdout */

putn(long dig) {
   char buffer[256];

   return fputs(ltoa(dig, buffer, 10), stdout);
   }
```

malloc

`#include <stdlib.h>`

`char *malloc(size_t size)`

malloc allocates a block of *size* bytes.

See the discussion in **Memory Management.**

RETURNS: *malloc* returns a pointer to the block, or `NULL` if it couldn't allocate the memory.

SEE ALSO: `calloc(), free(), freeall(), realloc()`

EXAMPLE:

```
/*
** strdup - copy a string
*/

char *strdup(char *str) {
   char *sav, *malloc();

   if(sav = malloc(strlen(str) + 1))
      strcpy(sav, str);
   return sav;
   }
```

memccpy

```
#include <string.h>

void *memccpy(void *dst, void *src,
                 char c, size_t n);
```

memccpy copies 0 or more bytes of *src* to *dst*, copying up to and including the first occurrence of *c* or until *n* bytes have been copied, whichever occurs first.

RETURNS: *memccpy* returns a pointer to the location of *c* in *dst*. Otherwise it returns NULL if *c* was not copied.

EXAMPLE:

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

char *memccpy(char *dst, char *src, char c,
              size_t n); {

    while(n--)
        if((*dst++ = *src++) == c)
            return dst - 1;
    return NULL;
    }
```

memchr

`#include <string.h>`

`void *memchr(void *str, char c, size_t n);`

memchr locates the first occurrence of *c* in the initial *n* characters of the object pointed to by *str* .

RETURNS: *memchr* returns a pointer to *c* , or `NULL` if *c* doesn't occur in the object.

EXAMPLE:

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

/* look for a string */

char *memchr(char *str, char c, size_t n) {

    while(n--)
        if(*str++ == c)
            return str - 1;
    return NULL;
    }
```

memcmp

```
#include <string.h>

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

memcmp compares the initial *n* characters of the object pointed to by *s2* to the the object pointed to by *s1* .

RETURNS: *memcmp* returns a value indicating the lexicographical relationship of *s1* to *s2* as follows:

Value	Meaning
<0	*s1* is less than *s2* .
0	*s1* is identical to *s2* .
>0	*s1* is greater than *s2* .

EXAMPLE:

```
#include <stddef.h>
#include <string.h>

/* compare two objects */

int memcmp(char *s1, char *s2, size_t n) {

    while(n--)
        if(*s1++ != *s2++)
            return *(s1 - 1) - *(s2 - 1);
    return 0;
    }
```

memcpy

```
#include <string.h>

void *memcpy(void *dst, void *src,
             size_t n);
```

memcpy copies *n* bytes of *src* to *dst* .

There is **no** test for overlap between *src* and *dst* .

RETURNS: *memcpy* returns the value of *dst* .

EXAMPLE:

```
#include <string.h>
#include <stddef.h>

char *memcpy(char *dst, char *src, size_t n) {
  char *beg = dst;

  while(n--)
    *dst++ = *src++;
  return beg;
  }
```

memicmp

```
#include <string.h>

int memicmp(void *s1, void *s2, size_t n);
```

memicmp compares the initial *n* characters of the object pointed to by *s2* to the the object pointed to by *s1*, without regard to the case of the characters.

RETURNS: *memicmp* returns a value indicating the case insensitive lexicographical relationship of *s1* to *s2* as follows:

Value	Meaning
<0	*s1* is less than *s2*.
0	*s1* is identical to *s2*.
>0	*s1* is greater than *s2*.

EXAMPLE:

```
#include <stddef.h>
#include <string.h>
#include <ctype.h>

#define TU(c) toupper(c)

/* compare two objects */

int memicmp(char *s1, char *s2, size_t n) {
    int c1, c2;

    while(n--)
        if((c1 = TU(*s1++)) != (c2 = TU(*s2++)))
            return c1 - c2;
    return 0;
    }
```

memmove

```
#include <string.h>

void *memmove(void *dst, void *src,
                  size_t n);
```

memmove copies *n* characters from *src* to *dst* .

memmove correctly copies overlapping objects.

RETURNS: *memmove* returns *dst* .

EXAMPLE:

```
#include <string.h>
#include <stddef.h>

char *memmove(char *dst, char *src, size_t n) {
    char *beg = dst;

    if(src + n > dst) {
        src += n;
        dst += n;
        while(n--)
            *--dst = *--src;
        }
    else
        while(n--)
            *dst++ = *src++;
    return beg;
    }
```

memset

`#include <stdlib.h>`

`void *memset(void *dst, char c, size_t n);`

memset sets the initial *n* bytes of *dst* to *c* .

RETURNS: *memset* returns *dst* .

EXAMPLE:

```
#include <string.h>
#include <stddef.h>

char * memset(char *dst, char c, size_t n) {
    char *beg = dst;

    while(n--)
        *dst++ = c;
    return beg;
    }
```

mkdir

```
int mkdir(pathName)
char pathName[];
```

mkdir creates a new directory *pathName*.

If drive and path components of *pathName* are specified, they must exist.

RETURNS: *mkdir* returns a 0 if the directory was created, or -1 on error.

SEE ALSO: `chdir(), rmdir()`

EXAMPLE:

```
/*
** create index file sub-directory
**
** form: path/name/index-name or data-name
**
** returns 0 == OK
*/

int icreat(name)
char *name;{
  char buf[128];

  getdir(0, buf);
  strcat(buf, "\\");
  strcat(buf, name);
  return mkdir(buf);
  }
```

modf

```
#include <math.h>

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

modf decomposes *value* into a positive fractional part and an integer part.

RETURNS: *modf* returns the positive fractional part and stores the integer part at **ipart*.

SEE ALSO: `frexp(), ldexp()`

EXAMPLE:

```
/*
** format the first n digits of val
*/

#include <math.h>

char *ndig(n, val, buf)
int n;
double val;
char *buf;{
   double i, f;
   int len;
   char wrk[32], *index();

   f = modf(val, &i);
   if(i)
      sprintf(buf, "%.0f", i);
   else
      buf[0] = '\0';
   sprintf(wrk, "%.16f", f);
   if((len = strcspn(buf, ".")) >= n){
      buf[n] = '\0';
   else
      strncat(&buf[len],index(wrk,'.')+1,n-len);
   return buf;
   }
```

open

```
#include <dos.h>

int sopen(name, mode)
char *name, mode;
```

open makes an existing file available for subsequent *read*, *write*, and *lseek* calls.

With any DOS release, *mode* can be:

VALUE	NAME	Action
0	READ	open the file for reading only
1	WRITE	open the file for writing only
2	READWRITE	open for reading and writing

With MSDOS version 3 and later, a sharing *mode* may be specified. The sharing modes are:

VALUE	NAME	Action
0x00	COMPAT	Compatibility mode, share with all other compatibility *opens*
0x10	DENYRW	deny read/write *opens*
0x20	DENYWR	deny write *opens*
0x30	DENYRD	deny read *opens*
0x40	DENYNO	deny no *opens*

RETURNS: *open* returns a handle that identifies the file in future file operations, or -1 if the file can't be opened.

NOTES: *open* can open the console ("CON"), the serial port ("AUX"), or the printer ("PRN").

EXAMPLE:

```
/*
** open, share with everybody
*/

#include <dos.h>

if((fh = open(name, READ + DENYNO)) == -1)
```

overlay
(Small Case Model)

```
int overlay_init(overlayFilename)
char *overlayFilename;

int overlay(overlayNumber)
int overlayNumber;

void overlay_close();

int  moverlay(overlayNumber)
int overlayNumber;
```

overlay_init must be called prior to the first *overlay* call and must be used when the `-V` option of BIND is used. *overlayFilename* contains the overlays. With DOS 2.0 and greater, the overlay file can be in the default directory of any directory listed in the PATH system parameter. Otherwise the file must be on the default drive or must explicitly contain the drive number, e.g. "B:X.OV".

overlay loads overlay *overlayNumber* as created by the `-V` option of BIND. It must be called before any reference or call to data or code in the overlay. Overlays are not automatically loaded by referencing a value in the overlay.

moverlay loads the indicated overlays created by the `-M` option of BIND. It works the same as the overlay function described above.

overlay_close closes all overlay files.

RETURNS: *overlay-init* returns -1 if the file could not be found. *overlay* and *moverlay* return -1 if *overlay_init* has not been called successfully, if the .OV file is bad, or if *overlayNumber* does not correspond to an existing overlay.

NOTE: When an overlay call is made, the functions in the previous overlays can no longer be called and the data associated with the last overlay is lost. If an uninitialized variable is referenced by both a module in the root and a module in an overlay, it is placed in the root. If a data item is initialized in a root module, it is placed in the root. If it is initialized in an overlay, it is placed in the overlay.

pow

```
#include <math.h>

double pow(x, y)
double x, y;
```

pow computes x^y.

RETURNS: *pow* returns the yth power of x, or 0.0 if x = 0.0 and y < 0.0, or if x < 0.0 and y is not an integer (*errno* is set to EDOM).

SEE ALSO: `exp(), log(), sqrt()`

EXAMPLE:

```
/*
** non-integer appx of pow()
*/

#include <math.h>

double nipow(x, y)
double x, y);{

  return exp(log(x) * y);
  }
```

printf

```
void printf(fcs [, arg ] ... );
char fcs[];
```

printf formats the output to the file *stdout.*

The format control string, *fcs*, contains both ordinary characters which are copied unchanged to the output, and conversion control strings which describe how each *arg* is to be formatted. Conversion control strings have the following format ([] enclose optional entries):

```
        %[-][width][parms]code
```

where the optional '-' specifies that the field is to be left justified — the default is right justification.

The optional *width* specifies the minimum field width in bytes. A '*' means that the width is specified by the next `int` *arg* in the calling sequence. A leading zero indicates that the field should be padded with zeroes instead of blanks. The field is not truncated if the width is too small.

Both *parms* and *code* depend upon the specific control string, as follows.

Character: `%[-][width]c`

```
printf("%c", "A")         → |A|
printf("%3c", "A")        → |  A|
printf("%*c", -3, "A")    → |A  |
```

String: `%[width][.precision]s`

precision specifies the maximum size of the string. An '*' means that the size is specified by the next `int` *arg* in the calling sequence. If the string is longer than the *precision* , then the string is truncated.

```
printf("%5s", "abcdefgh")    → |abcdefgh|
printf("%-5.3s, "abcdefgh")  → |abc  |
printf("%5.3s, "abcdefgh")   → |  abc|
```

printf

Signed Integer: `%[-][sign][width][l]d`

A leading minus sign '-' is automatically output for negative numbers. If the optional *sign* is a `'+'`, a leading plus sign is output for positive numbers; a space outputs a blank for positive numbers.

The optional *l* (lowercase 'L') specifies that the corresponding *arg* is a `long`.

```
printf("%d", -45)      → |-45|
printf("%+d", 45)      → |+45|
printf("% ld", 45L)    → | 45|
printf("%0*d", 3, 45)  → |045|
```

Unsigned Integer: `%[-][#][width][l]code`

# specifies that a leading '0' is output for octal numbers, and a leading '0x' is output for hexadecimal numbers.

code is 'u' for decimal format, 'o' for octal format, and 'x' for hexadecimal format.

```
printf("%u", 255)   → |255|
printf("%o", 255)   → |377|
printf("%#x", 255)  → |0xFF|
```

Floating Point: `%[-][sign][#][.precision]code`

specifies that trailing zeroes are to be output, and that a decimal point is output, even for zero precision.

precision specifies the number of digits output after the decimal point for *code* 'e' and 'f', or the number of significant digits for *code* 'g'. An '*' means that the number of digits is specified by the next `int` *arg* in the calling sequence. Truncation causes rounding. The default for *precision* is 6.

code is 'e' for [-]*d.ddddd* E[-]*dd* format, 'f' for [-]*ddd.ddd* format, and 'g' for the shorter of 'e' or 'f' formats.

printf

```
printf("%f", 1234.56789)    → |1234.567890|
printf("%.1f", 1234.56789)  → |1234.6|
printf("%.3e", 1234.56789)  → |1.235E03|
printf("%g", 1234.56789)    → |1234.57|
```

Literal %: `%%` `printf("%5.2f%%", 99.44) → |99.44%|`

NOTE The maximum *printf* and *fprintf* output is 256 bytes. If you need more use *sprintf* followed by `puts()`.

putc, putchar

```
#include <stdio.h>

int putc(ch, fp)
char ch;
FILE *fp;
```

putc writes *ch* to the file *fp* . *fp* must be open.

putchar writes *ch* to *stdout* .

Linefeed ('\n') is converted to carriage return - linefeed ('\r\n'). Output will stop if CTRL-S is entered, and resume when any other key is pressed. Each output will check for a CTRL-C entry and terminate the program if it was pressed.

RETURNS: *putc* and *putchar* return *ch* , or EOF on error.

SEE ALSO: `printf(), write()`

NOTE: *putchar* is a function rather than a macro.

putenv

```
int putenv(key, newValue)
char *key, newValue[];
```

putenv changes the value associated with *key* in the environment to *newValue*. *newValue* is a `NULL`-terminated, possibly empty, string.

putenv searches the DOS environment for an entry of the form

key =oldValue

If *key* is found, *newValue* replaces *oldValue*. If *key* is not found, then a new *key=newValue* entry is inserted in the environment.

RETURNS: *putenv* returns 0, or -1 on error.

NOTE: *key* is terminated by the '=' character, so

```
PATH=C:\
```

and

```
PATH =C:\
```

are different environment entries.

SEE ALSO: `getenv()`

EXAMPLE:

```
/*
** update parents cycle number
*/

char buf[16];

getenv("CYCLE", buf);
sprintf(buf, "%d", atoi(buf) + 1);
putenv("CYCLE", buf);
```

puts

```
#include <stdio.h>

int puts(buf)
char buf[];
```

puts copies the null terminated string *buf* to the console (*stdout*).

On output, linefeed ('`\n`') is converted to carriage return - linefeed ('`\r\n`'). Output will stop if CTRL-S is entered and resume when any other key is pressed.

puts will check for a CTRL-C entry and terminate the program if one occurred.

RETURNS:	*puts* returns a -1 on error.
SEE ALSO:	`fprintf(), fwrite()`
NOTE:	*puts* doesn't append a newline.

putw

```
#include <stdio.h>

int putw(w, fp)
int w;
FILE *fp;
```

putw writes the `int` *w* to the file *fp* . *fp* must be open.

RETURNS: *putw* returns *w*, or -1 on error.

SEE ALSO: `printf()`, `write()`

NOTE: There is no way to distinguish the return from `putw(-1, fp)` from an error.

qsort

```
void qsort(array, num, width, compare)
char array[];
int num, width, (*compare)();
```

qsort is an implementation of C. A. R. Hoare's *quicker-sort* algorithm. It sorts an *array* of *num* elements, each *width* bytes wide. *compare* is called with two arguments (pointers to the two elements being compared), and returns an integer less than, equal to, or greater than zero accordingly as the first argument is less than, equal to, or greater than the second argument.

RETURNS: *qsort* does not return any value.

NOTE: The usual function for *compare* is `strcmp()`. If you are sorting anything other than strings, the following may serve as a model:

```
int compare(left, right)
int *left, *right; {
  return *left - *right;
  }

#define TCARD    1024
#define ISIZE    sizeof(int)

int itab[TCARD];

  qsort(itab, TCARD, ISIZE, compare);
```

Remember that `int`, `long`, `float`, and `double` values are stored with their low-order bytes first. Thus string comparisons (i.e., `strcmp()`) may not produce the expected results.

rand

```
int rand();
```

rand computes the next pseudo-random number in the range from 0 to $2^{15} - 1$. Repeated runs of the program produce identical sequences of pseudo-random numbers.

If you wish different sequences of numbers, or to restart the sequence within a run, call *srand* to initialize the pseudo-random number generator.

RETURNS: *rand* returns the next pseudo-random number. There are no error codes.

SEE ALSO: `frand(), srand()`

EXAMPLE:

```
/*
** flip a coin
*/

int coinFilp(){

  return rand() > 0x4000;
  }
```

read

```
int read(handle, buf, count)
int handle;
char *buf;
unsigned count;
```

read reads *count* bytes into *buf* from the file *fp*.

read starts reading from the current position of *fp*. After the read, the current position has advanced *count* bytes or is at EOF.

RETURNS: *read* returns the number of bytes actually read (which may be less than *count* if end-of-file is reached), or -1 if an error occurred.

NOTE: There is no way to distinguish a normal *read* of 0xFFFF bytes from an error.

SEE ALSO: `fgetc(), fgets(), scanf()`

EXAMPLE:

```
/*
** Small Case block transfer
*/

xfer(ih, oh)
int ih, oh;{
  char *buf, *_memory();
  unsigned size, amt;

  freeall(256); /* min stack */
  buf = _memory() + 1; /* point to size */
  size = (unsigned *)buf & 0xF800; /* % 2K */
  buf += 2; /* point to buffer */
  do {
    if((amt = read(ih, buf, size)) = -1){
      puts("xfer: read error);
      exit(1);
      }
    write(oh, buf, amt);
    } while(amt == size);
  close(oh);
  }
```

realloc

```
char *realloc(op, size)
char *op;
unsigned size;
```

realloc changes the size of the block at *op* to *size* bytes.

RETURNS: *realloc* returns a pointer to the (possibly moved) block, or 0 if it couldn't allocate the memory. If *realloc* returns 0, the original block at *op* is still allocated and useable.

See the description of *malloc* for a discussion of the memory allocation area.

SEE ALSO: `calloc(), free(), freeall(), malloc()`

EXAMPLE:

```
/*
** enlarge saved string
*/

char *savcat(str, add)
char *str, *add);{
  unsigned size;
  char *new;

  size = strlen(str) + strlen(add) + 1;
  if(new = realloc(str, size))
    strcat(new, add);
  return new;
  }
```

remove

```
#include <stdio.h>

int remove(char *pathname);
```

remove deletes the file specified by *pathname* .

RETURNS: *remove* returns 0 if the file was deleted, -1 otherwise.

EXAMPLE:

```
#include <stdio.h>

int mustDelete(char *name) {

  if(remove(name)) {
    printf("can't remove %s\n", name);
    abort();
    }
  }
```

Intentionally blank

rename

```
int rename(oldFile, newFile)
char *oldFile, *newFile;
```

rename changes the file name *oldFile* to *newFile.*

Under DOS 2 and later, *oldFile* may contain a path specification.

RETURNS: *rename* returns -1 if *oldFile* is open or if an error is detected.

EXAMPLE:

```
/*
** create .BAK file
*/

makeBAK(old)
char *old;{
   char *suf, new[128];


strcpy(new, old);
if((suf = rindex(new, '.'))
   && suf > old && *(suf - 1) != '.')
      *suf = '\0';
strcat(new, ".BAK");
return rename(old, new);
}
```

rewind

```
#include <stdio.h>

long rewind(fp);
FILE *fp;
```

rewind is the same as `fseek(fp,0L,0)` — it seeks to the begining of the file *fp* .

RETURNS: *rewind* returns 0L, or -1 in case of error.

SEE ALSO: `fseek(), ftell()`

rindex

```
char *rindex(src, ch);
char src[], ch;
```

rindex locates the last occurence of *ch* in *src*.

RETURNS: *rindex* returns a pointer to the last occurence of *ch* in *src*, or 0 if *ch* isn't in *src*.

SEE ALSO: `index()`

EXAMPLE: See ***rename***.

rmdir

```
int rmdir(pathname)
char pathname[];
```

rmdir deletes the directory *pathname.*

pathname must be empty and must not be the current working directory or the root directory.

RETURNS: *rmdir* returns 0 if the directory was deleted, -1 otherwise.

SEE ALSO: `chdir(), mkdir()`

scanf

```
int scanf(fcs [, ptr ] ... )
char fcs[];
```

scanf reads from *stdin* . The format control string, *fcs*, contains: blanks or tabs, which match optional whitespace (blanks, tabs, newlines, formfeeds, and vertical tabs) in the input; a non-'%' character which must match the next character in the input, and conversion control strings which describe the type and format of each **ptr*. Conversion control strings have the following format ([] enclose optional entries):

```
%[*][width][parms]code
```

where: * indicates that the field should be skipped and not assigned to a **ptr* and *width* specifies the maximum field size in bytes. Both *parms* and *code* are described below. The examples have the following form:

| input string | → function call → result

Character: `%[*][width]c`
String: `%[*][width]s`

width specifies the number of characters to be read into the array at **ptr* . The default is 1. 'c' whitespace is not skipped, 's' whitespace is skipped.

```
|  abc| → scanf("%3c", buf) → |  a|
|  abc| → scanf("%3s", buf) → |abc|
```

Integer: `%[*][width][size]code`

size equal to 'l' (lowercase 'L') specifies that **ptr* point to a `long`, an 'h' specifies a `short int` .

code is one of: 'd' — signed decimal format, 'u' — unsigned decimal format, 'o' — unsigned octal, and 'x' — unsigned hexadecimal.

```
|  FF  | → scanf("%x", &hex)  →  255
| 377  | → scanf("%o", &oct)  →  255
```

scanf

Floating Point: `%[*][width][size]code`

size equal to 'l' (lowercase 'L') specifies that **ptr* points to a `double` rather than a `float`.

code can be either 'e', 'f', or 'g' — they all indicate floating point.

```
| 123.45 | → scanf("%f", %flt)         → 123.45
| 123.45 | → scanf("%4lf%d", &d, &i) → 123.0  45
```

Scanset: `%[*][width]scanset`

scanset is specified by a sequence of characters enclosed by brackets '[' ']'. It reads a string, including the terminating null character. Leading whitespace <u>is not</u> skipped.

```
|123 ABC| → scanf("%[123 ]", str) → |123 |
```

A range of contiguous characters can be specified by the first and last element of the range, separated by a '–'.

```
|123 ABC| → scanf("%[1-3 ]", str) → |123 |
```

If the first element of *scanset* is a `'^'`, then all characters *except* those specified will be read.

```
|123 ABC| → scanf("%[^A-C]", str) → |123 |
```

To specify '–' or '^' in a *scanset*, specify it as the first element. Thus to read an integer, skip any interviening garbage, and read another integer

```
scanf("%d%*[^-+0-9]%d", &dig1, &dig2);
```

RETURNS: *scanf* returns the number of items successfully scanned, or EOF if a CTRL-Z was read.

SEE ALSO: `fscanf(), printf(), sscanf()`

scr_aputs

```
void scr_aputs(string, attr);
char *string, attr;
```

scr_aputs writes string *str* to the display with attribute *attr*. '\r' moves to the begining of the line, and '\n' moves to the next line. Moving off the bottom line causes scrolling.

attr is defined in the IBM PC Technical Reference Manual.

RETURNS: *sacr_aputs* returns no value.

NOTE: *scr_aputs* is in the file PCIO.A. It is for use on machines that support the INT 10H interface.

EXAMPLE:

```
/*
** write text in reverse video
*/

highLight(text)
char *text;{

   scr_aputs(text, 0x70);
   }`
```

scr_ci, scr_co, scr_csts

```
char scr_ci();
void scr_co(ch);
char scr_csts();
```

scr_ci reads the keyboard like `ci()` but uses its own translation table for command characters. The table is in the files CONFIG.C and PCIO.A.

scr_csts tests for a character in the input queue, and if one is found, reads it. The character is not retained.

scr_co writes a character to the display like `co()`.

RETURNS: *scr_ci* returns the next character from the input queue. *scr_csts* returns the next character from the queue, or 0 if no character is available. *scr_co* returns no value.

NOTE: All the functions are in the files PCIO.A and CONFIG.C

scr_ci and *scr_csts* use INT 16H in the file PCIO.A.

SEE ALSO: `scr_setmode(), scr_setup()`

EXAMPLE:

```
/*
** empty keyboard queue
**
** scr_csts() reads the character from the
**            input queue - csts() doesn't
*/

kbdFlush(){
  while(scr_csts())
      ;
  }
```

scr_clr, scr_clrl, scr_cls

```
void scr_clr();
void scr_clrl();
void scr_cls();
```

scr_clr erases the entire screen.

scr_clrl erases everything from the cursor location to the end of the line.

scr_cls erases everything from the cursor location to the end of the screen.

RETURNS: There are no values returned.

NOTE: All the functions are in the file PCIO.A.

SEE ALSO: `scr_setmode(), scr_setup()`

EXAMPLE:

```
/* menu processing */

struct _menu {
     int trow, tcol;
     char *text, attrib;
     int rrow, rcol;
     char *response;
     int (*valid)();
     };

doMenu(mp)
struct _menu mp[];{

   scr_clr():
   while(mp->trow != -1){
      scr_rowcol(mp->trow, mp->tcol);
      scr_aputs(mp->text, mp->attrib);
      scr_rowcol(mp->rrow, mp->rcol);
      while(!(*mp->valid)(mp->response)){
         scr_rowcol(mp->rrow, mp->rcol);
         scr_clrl();
      }
   }
```

scr_cursoff, scr_curson

```
void scr_cursoff();
void scr_curson();
```

scr_cursoff turns the cursor off; *scr_curson* turns it back on.

RETURNS: The functions return no values.

NOTE: Both functions are in the file PCIO.A.

scr_setup must be called prior to calling these functions.

SEE ALSO: `scr_setmode(), scr_setup()`

EXAMPLE:

```
/*
** display line count
*/

scr_setup();
scr_cursoff();
scr_clr();
lineno = 0;
while(fgets(buf, sizeof(buf), fp))
     printf("\r%u", lineno++);
scr_curson();
```

scr_rowcol

```
void scr_rowcol(trow, tcol)
int trow, tcol;
```

scr_rowcol moves the cursor to row *trow* and column *tcol*.

RETURNS: *scr_rowcol* returns no value.

NOTE: *scr_rowcol* is in the file PCIO.A.

SEE ALSO: `scr_setmode(), scr_setup()`

EXAMPLE:

```
/* menu processing */

struct _menu {
     int trow, tcol;
     char *text, attrib;
     int rrow, rcol;
     char *response;
     int (*valid)();
     };

doMenu(mp)
struct _menu mp[];{

   scr_clr():
   while(mp->trow != -1){
      scr_rowcol(mp->trow, mp->tcol);
      scr_aputs(mp->text, mp->attrib);
      scr_rowcol(mp->rrow, mp->rcol);
      while(!(*mp->valid)(mp->response)){
         scr_rowcol(mp->rrow, mp->rcol);
         scr_clrl();
      }
   }
```

scr_scdn, scr_scrdn

`void scr_scdn();`

`void scr_scrdn(lines, fr, fc, tr, tc);`
`int lines, fr, fc, tr, tc;`

scr_scdn scrolls the screen down one line, but leaves the top two lines alone.

scr_scrdn scrolls the given area down *lines*. The area is defined by the upper-left location (*fr*, *fc*) and the lower-right location (*tr*, *tc*).

RETURNS: The functions return no values.

NOTE Both functions are in the file PCIO.A.

SEE ALSO: `scr_setmode(), scr_setup()`

EXAMPLE:

```
/*
** scroll a window
*/

struct _win {
     char ul_row, ul_col;
     char lr_row, lr_col;
     char *name;
     char attr;
     } _window[MAXWIN];

wscroll(window, lines)
int window, lines;{
  struct _win *wp = &_window[window];

  if(lines < 0)
     scr_scrdn(-lines, wp->ul_row, wp->ul_col,
                       wp->lr_row, wp->lr_col);
  else
     scr_scrup(lines,  wp->ul_row, wp->ul_col,
                       wp->lr_row, wp->lr_col);
  }
```

scr_scup, scr_scrup

```
void scr_scup();

void scr_scrup(lines, fr, fc, tr, tc);
int lines, fr, fc, tr, tc;
```

scr_scup scrolls the screen up one line, but leaves the top two lines alone.

scr_scrup scrolls the given area up *lines*. The area is defined by the the upper-left location (*fr*, *fc*) and the lower-right location (*tr*, *tc*).

RETURNS: The functions return no values.

NOTE Both functions are in the file PCIO.A.

SEE ALSO: `scr_setmode(), scr_setup()`

EXAMPLE:

```
/*
** scroll a window
*/

struct _win {
    char ul_row, ul_col;
    char lr_row, lr_col;
    char *name;
    char attr;
    } _window[MAXWIN];

wscroll(window, lines)
int window, lines;{
  struct _win *wp = &_window[window];

  if(lines < 0)
    scr_scrdn(-lines, wp->ul_row, wp->ul_col,
                      wp->lr_row, wp->lr_col);
  else
    scr_scrup(lines,  wp->ul_row, wp->ul_col,
                      wp->lr_row, wp->lr_col);
  }
```

scr_setmode, scr_setup

```
void scr_setmode(newMode)
char newMode;

void scr_setup();
```

scr_setmode sets the mode of the graphics card. *newMode* must be between 0 and 6. See the Note below.

scr_setup must be called prior to any of the screen routines if the screen is currently in 80 column mode or if *scr_curson* with a monochrome display is used. This routine sets the value of the global variables described in the Note below.

RETURNS: The functions return no values.

NOTE: Both functions are in the file PCIO.A.

scr_setmode and *scr_setup* manage the following global data.

```
char scr_cols; /* number of character positions */
char scr_rows; /* number of lines */
char scr_mode; /* current screen mode:
      0 = 40 col. BW
      1 = 40 col. color
      2 = 80 col. BW
      3 = 80 col. color
      4 = 320 x 200 color graphics
      5 = 320 x 200 BW graphics
      6 = 640 x 200 BW graphics
      7 = 80 col. BW    */
char scr_page; /* current active display page */
```

scr_sinp

`char scr_sinp();`

scr_sinp reads the character at the current cursor location.

RETURNS: *scr_sinp* returns a character.

NOTE: *scr_sinp* is in the file PCIO.A.

SEE ALSO: `scr_setmode(), scr_setup()`

EXAMPLE:

```
/*
** read numeric field from screen
**
** no test that col + len on screen
*/

double readnAt(row, col, len)
char row, col, len;{
   char data[80], dp = data;
   double atof();

   while(n--){
      scr_rowcol(row, col++);
      *dp++ = scr_sinp();
      }
   *dp = '\0';
   return atof(data);
   }
```

setjmp

```
#include <setjmp.h>

int setjmp(env)
jmp_buf env;
```

jmp_buf is defined in *<setjmp.h>* . It creates an environment used by *setjmp* for future use by *longjmp* .

setjmp saves the environment in *env* .

RETURNS: *setjmp* returns 0 after saving the environment, or a non-zero value as the result of a *longjmp* call.

SEE ALSO: `longjmp()`

NOTE: *env* can be specified as zero for compatibility with previous releases. There can be only one "zero" *env* active at any time.

If the environment stored in *env* points into an overlay area, then the overlay that called *setjmp* must be resident when *longjmp* is called — if another overlay is resident, then strange things will happen. It is best to call *setjmp* from the root.

EXAMPLE:

```
/*
** error handler
*/

#include <setjmp.h>
#include <stdio.h>

jmp_buf err;

#define error(n) longjmp(err, n)

main(){
   int code;

   if(code = setjmp(err))
      fprintf(stderr, "\nerror[%d]\n", code);
   process();
   }
```

sin

```
#include <math.h>

double sin(x)
double x;
```

sin computes the sine of its radian argument *x*. The meaningfulness of the result depends upon the magnitude of the argument.

RETURNS: *sin* returns the sine of its argument. *sin* doesn't set any error codes.

SEE ALSO: `acos(), asin(), atan(), cos(), tan()`

sprintf

```
void sprintf( buf, fcs, [, arg ] ...)
char buf[], fcs[];
```

sprintf formats the output into *buf*, which must be large enough to contain the output.

The format control string, *fcs*, contains both ordinary characters which are copied unchanged to the output, and conversion control substrings which describe how each *arg* is to be formatted. *fcs* is described in *printf*.

RETURNS: *sprintf* doesn't return a value.

SEE ALSO: `fprintf(), printf(), scanf()`

EXAMPLE:

```
/*
** format an array of longs as a long string
** (>256 bytes) and send to file fh
*/

int lsfmt(da, n, fp)
double *da;
int n, fh;{
  char fcs[256], buf[4096];
  int oldsp, newsp, i;
  double *dp;

  for(i = 0, fcs[0] = '\0'; i < n; i++)
    strcat(fcs, " %.16g");
  oldsp = _showsp();
  _setsp(newsp = oldsp - n * sizeof(double));
  for(i = 0, dp = newsp; i < n; i++)
    *dp++ = *da++; /*copy to stk for sprintf*/
  sprintf(buf, fcs);
  _setsp(oldsp);
  strcat(buf, "\n");
  return write(fh, buf, strlen(buf));
  }
```

sqrt

```
#include <math.h>
```

```
double sqrt(x)
double x;
```

sqrt computes the square root of *x*.

RETURNS: *sqrt* returns the computed square root. If *x* is < 0.0, *sqrt* returns 0.0 and sets *errno* to **EDOM**.

SEE ALSO: `exp(), log(), pow()`

srand

```
void srand(seed)
int seed;
```

srand sets the seed for the random number generator to *seed* .

RETURNS: *srand* returns no value.

SEE ALSO: `frand(), rand()`

NOTE: The internal form of the seed for the generator is a `long`. *srand* only sets the low-order word, so the generator cannot be restarted to its initial value. Solution: seed the generator with your own integer before any calls to *rand* or *frand* .

EXAMPLE:

```
/*
** keep simulating the same pattern
*/

puts("Starting...");
do {
   srand(1);  /* init generator */
   simulate();
   puts("One more time? (y/n)...");
   } while(toupper(getchar()) == 'Y');
```

sscanf

```
int sscanf(buf, fcs [, ptr ] ...)
char buf[], fcs[];
```

sscanf reads from the string *buf*, assembles data under the specification of *fcs*, and stores the data at **ptr*.

The format control string, *fcs*, is described in *scanf*.

RETURNS: *sscanf* returns the number of fields scanned and assigned. A return of zero means no fields were converted.

NOTE: Use *sscanf* when the input line being scaned exceeds 256 bytes.

SEE ALSO: `fscanf(), scanf()`

EXAMPLE:

```
/*
** scan a very long line of doubles that begin
**  with a space from file fp
** return number scanned
*/

lscanf(fp, da)
FILE *fp;
double *da;{
   int i, n = 0;
   char buf[4096], *bp = buf;

   if(fgets(buf, 4096, fp) == NULL)
      return 0;
   while(i = sscanf(bp, "%d %d %d %d",
                        da, da+1, da+2, da+3)){
      n += i;
      da += i;
      while(i--)
         bp = index(bp + 1, 0x20);
      if(i < 4)
         break;
      }
   return n;
   }
```

strcat

```
char *strcat(dst, src)
char *dst, src[];
```

strcat appends a copy of *src* (including the terminating `'\0'`) to the end of *dst* .

There is no test for overflow.

RETURNS: *strcat* returns a pointer to the concatenated string, *dst*.

SEE ALSO: `strncat()`

EXAMPLE:

```
/*
** concatenate src to the end of dst.
** dst is assumed to be large enough
**     to hold both dst and src
**
** return dst
*/

char *strcat(dst, src)
char *dst, *src;{
   char *ret = dst;

   while(*dst++)
      ;
   dst--;
   while(*dst++ = *src++)
      ;
   return ret;
   }
```

strchr

`#include <string.h>`

`char *strchr(char *str, char c);`

strchr locates the first occurrence of *c* in the string *str* .

The terminating null character is considered to be part of the string.

RETURNS: *strchr* returns a pointer to *c* , or NULL if *c* doesn't occur in *str* .

EXAMPLE:

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

char *strchr(char *str, char c) {
    int n;

    n = strlen(str) + 1; /* include null */
    while(n--)
        if(*str++ == c)
            return str - 1;
    return NULL;
    }
```

Intentionally blank

strcmp

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

strcmp compares the the contents of *s1* with the contents of *s2*. The comparison stops when a mismatch occurs or when the end-of-string character (`'\0'`) is encountered.

RETURNS: *strcmp* returns a value indicating the result of the comparison.

Value	Comparison
+1	*s1* is lexically greater than than *s2*
0	*s1* is lexically equal to *s2*
-1	*s1* is lexically less than *s2*

SEE ALSO: `strcmpi(), strncmp()`

EXAMPLE:

```
/*
** comapre two strings
*/

int strcmp(s1, s2)
char *s1, *s2;{

    if(s1 == s2)
        return 0;
    while(*s1 == *s2++)
        if(*s1++ == '\0')
            return 0;
    return *s1 - *(--s2);
```

strcmpi

```
int strcmpi(s1, s2)
char *s1, *s2;
```

strcmpi compares the the contents of *s1* with the contents of *s2* without regard for case — the upper and lower-case forms of a character are equivalent. The comparison stops when a mismatch occurs or when the end-of- string character (`'\0'`) is encountered.

RETURNS: *strcmpi* returns a value indicating the result of the comparison.

Value	Comparison
+1	*s1* is lexically greater than than *s2*
0	*s1* is lexically equal to *s2*
-1	*s1* is lexically less than *s2*

SEE ALSO: `strcmp(), strncmp()`

EXAMPLE:

```
/*
** comapre two strings without regard to case
*/

int strcmpi(s1, s2)
char *s1, *s2;{

    if(s1 == s2)
        return 0;
    while(tolower(*s1) == tolower(*s2++))
        if(*s1++ == '\0')
            return 0;
    return tolower(*s1) - tolower(*(--s2));
```

strcpy

```
char *strcpy(dst, src)
char *dst, src[];
```

strcpy copies *src* to *dst*, stopping after the terminating `'\0'` has been transferred.

There is no test for overflow.

RETURNS: *strcpy* returns a pointer to the copied string, *dst*.

SEE ALSO: `strcat(), strncpy()`

EXAMPLE:

```
/*
** copy src to dst
*/

char *strcpy(dst, src)
char *dst, *src;{
  char *ret = dst;

  while(*dst++ = *src++)
     ;
  return ret;
  }
```

strcspn

```
int strcspn(s1, s2)
char s1[], s2[];
```

strcspn searches *s1* for any of the characters in *s2*. The index of the first character found is also the length of the begining substring of *s1* that consists entirely of characters not in *s2*. Terminating `'\0'` characters are not part of the search.

RETURNS: *strcspn* returns the index of the first character of *s1* that is in *s2*..

SEE ALSO: `strspn()`

EXAMPLE:

```
/*
** return the number of characters in the
** begining of s1 that are NOT in s2
*/

int strcspn(s1, s2)
char *s1, *s2;{
  char *s1p, *s2p;

  for(s1p = s1; *s1p; s1p++){
    for(s2p = s2; *s2p && *s2p != *s1p; s2p++)
      ;
    if(*s2p)
      break;
    }
  return s1p - s1;
  }
```

strdup

```
char *strdup(src)
char src[];
```

strdup allocates storage space via ***malloc*** for a copy of *str*, and copies *str* (including the terminating `'\0'`) into that space.

RETURNS: *strdup* returns a pointer to the allocated area, or `NULL` if the area couldn't be allocated.

SEE ALSO: `malloc(), strcpy()`

EXAMPLE:

```
/*
** save a copy of a string
*/

char *strdup(str)
char *str;{
   char *sav, *malloc();

   if(sav = malloc(strlen(str) + 1))
      strcpy(sav, str);
   return sav;
   }
```

stricmp

```
#include <string.h>

int stricmp(char *s1, char *s2);
```

stricmp compares the string pointed to by *s2* to the string pointed to by *s1* , without regard to the case of the characters.

RETURNS: *stricmp* returns a value indicating the case insensitive lexicographical relationship of *s1* to *s2* as follows:

Value	Meaning
<0	*s1* is less than *s2* .
0	*s1* is identical to *s2* .
>0	*s1* is greater than *s2* .

EXAMPLE:

```
#include <stddef.h>
#include <string.h>
#include <ctype.h>

#define TU(c) toupper(c)


int stricmp(char *s1, char *s2) {
    int n1, n2, c1, c2;

    n1 = strlen(s1);
    n2 = strlen(s2);
    if(n1 > n2)
        n1 = n2;
    while(n1--)
        if((c1 = TU(*s1++)) != (c2 = TU(*s2++)))
            return c1 - c2;
    return TU(*s1) - TU(*s2);
    }
```

strlen

`int strlen(char src[])`

strlen counts the number of characters in *src* , excluding the terminating `'\0'`.

RETURNS: *strlen* returns the length. There are no error codes.

EXAMPLE:

```
/*
** return the string length
*/

int strlen(str)
char *str;{
    char *beg = str + 1;

    while(*str++)
        ;
    return str - beg;
    }
```

Intentionally blank

strlwr

```
int strlwr(src)
char src[];
```

strlwr converts any upper-case characters in *src* to lower-case.

RETURNS: *strlwr* returns a pointer to the converted string, *src*.

SEE ALSO: `strupr()`

EXAMPLE:

```
/*
** convert string to lower case
*/

char *strlwr(str)
char *str;{
    char *ret = str;

    for(; *str; str++)
        *str = tolower(*str);
    return ret;
    }
```

strncat

```
char *strncat(dst, src, max)
char *dst, src[];
int max;
```

strncat appends, at most, *max* bytes of *src* to the end of *dst* .

RETURNS: *strncat* returns a pointer to the concatenated string, *dst*.

SEE ALSO: `strcat()`

EXAMPLE:

```
/*
** concatenate at most n bytes of src to dst;
**
** return dst
*/

char *strncat(dst, src, n)
char *dst, *src;
int n;{
    char *ret = dst;

    while(*dst++)
        ;
    dst--;
    while(*dst++ = *src++){
        if(n--)
            continue;
        *(--dst) = '\0';
        break;
        }
    return ret;
    }
```

strncmp

```
int strncmp(s1, s2, max)
char *s1, *s2;
int max;
```

strncmp compares at most, *max* bytes of the two strings *s1* and *s2*.

RETURNS: *strncmp* returns a value indicating the result of the comparison.

Value	Comparison
+1	*s1* is lexically greater than than *s2*
0	*s1* is lexically equal to *s2*
-1	*s1* is lexically less than *s2*

SEE ALSO: `strcmp(), strcmpi()`

EXAMPLE:

```
/*
** compare at most n bytes of two strings
*/

int strncmp(s1, s2, n)
char *s1, *s2;
int n;{

    if(s1 == s2)
        return 0;
    while(n-- && *s1 == *s2++)
        if(*s1++ == '\0')
            return 0;
    return n == 0 ? 0 : *s1 - *(--s2);
    }
```

strncpy

```
char *strncpy(dst, src, len)
char *dst, src[];
int len;
```

strncpy copies exactly *len* bytes of *src* to *dst*, truncating or padding with `'\0'` as required; *dst* may not be null-terminated if `strlen(src) >= len`.

RETURNS: *strncpy* returns a pointer to the copied string, *dst*.

NOTE: The returned string is not terminated with a `'\0'` if *n* is greater than or equal to `strlen(src)`.

SEE ALSO: `strcpy()`

EXAMPLE:

```
/*
** copy src to dst -- truncate or pad with 0
** so that exactly n bytes are copied
**
** return dst
*/

char *strncpy(dst, src, n)
char *dst, *src;
int n;{
    char *ret = dst;

    while(n--)
        if((*dst++ = *src++) == '\0'){
            while(n--)
                *dst++ = '\0';
            break;
            }
    return ret;
    }
```

strpbrk

```
char *strprbk(s1, s2)
char s1[], s2[];
```

strpbrk searches *s1* for any character from *s2*. The terminating `'\0'` characters are not included in the search.

RETURNS: *strpbrk* returns a pointer to the first character found, or `NULL` if the strings have no character in common.

SEE ALSO: `index(), rindex()`

EXAMPLE:

```
/*
** return pointer to first char in s1 that is
** also in s2 -- NULL otherwise
*/

#define NULL (char *)0

char *strpbrk(s1, s2)
char *s1, *s2;{
   char *s2p;

   do {
      for(s2p = s2; *s2p && *s2p != *s1; s2p++)
         ;
      if(*s2p)
         return s1;
      } while(*s1++);
   return NULL;
   }
```

strrchr

`#include <string.h>`

`char *strrchr(char *str, char c);`

strchr locates the last occurrence of *c* in the string *str* .

The terminating null character is considered to be part of the string.

RETURNS: *strrchr* returns a pointer to *c* , or NULL if *c* doesn't occur in *str* .

EXAMPLE:

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

char *strrchr(char *str, char c) {
    char *beg = str;

    while(*str++)
        ;
    while(--str >= beg)
        if(*str == c)
            return str;
    return NULL;
    }
```

strrev

```
char *strrev(src)
char src[];
```

strrev reverses the order of characters in *src*.

RETURNS: *strrev* returns a pointer to the altered string, *src*.

SEE ALSO: `strcpy(), strset()`

EXAMPLE:

```
/*
** reverse elements of a string
*/

char *strrev(src)
char *src;{
    char *beg, *end, ch;

    beg = end = src;
    while(*end++)
        ;
    end -= 2;  /* point to last char */
    while(end > beg){
        ch = *end;
        *end-- = *beg;
        *beg++ = ch;
        }
    return src;
    }
```

Intentionally blank

strset

```
char *strset(dst, ch)
char dst[], ch;
```

strset sets all of the characters of *dst* to *ch*.

RETURNS: *strset* returns a pointer to the altered string, *dst*.

SEE ALSO: `_setmem()`

EXAMPLE:

```
/*
** fill dst with ch
*/

char *strset(dst, ch)
char *dst, ch;{
    char *ret = dst;

    while(*dst)
        *dst++ = ch;
    return ret;
    }
```

strspn

```
char *strspn(s1, s2)
char *s1, *s2;
```

strspn searches *s1* for a character that is not a member of *s2*. The index of the first character in *s1* not in *s2* is also the length of the beginning substring of *s1* that consists entirely of characters in *s2*. Terminating `'\0'` characters are not part of the search.

RETURNS: *strspn* returns the index of the first character of *s1* that is not in *s2*..

SEE ALSO: `strcspn()`

EXAMPLE:

```
/*
** return length of initial substring of s1
** made up solely from members of s2
/*

int strspn(s1, s2)
char *s1, *s2;{
   char *s1p, *s2p;

   for(s1p = s1; *s1p; s1p++){
      for(s2p = s2; *s2p && *s2p != *s1p; s2p++)
         ;
      if(*s2p == '\0')
         break;
      }
   return s1p - s1;
   }
```

strstr

`#include <string.h>`

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

strstr locates the first occurrence of *s2* (excluding the terminating null character) in *s1* .

RETURNS: *strstr* returns a pointer to the occurrence of *s2* , `NULL` otherwise.

EXAMPLE:

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

char *strstr(char *s1, char *s2) {
    int n;

    if(n = strlen(s2))
        while(s1 = strchr(s1, *s2)) {
            if(memcmp(s1, s2, n) == 0)
                return s1;
            s1++;
            }
    return NULL;
    }
```

strtod — strtol

```
#include <stdlib.h>

double strtod(char s[], char **eptr);

long strtol(char s[], char **eptr, int b);
```

strtod and *strtol* convert *s* to a `double` or a `long`, respectively. The conversion continues until the first character which cannot be converted is found. If *eptr* is not `NULL`, a pointer to the terminating character is put in **eptr*.

If *b* is between 2 and 36, it is used as the number base for the conversion. If *b* is 0, then the initial digits of *s* are used to determine the base: if `s[0]` is `'0'` and `s[1]` is an octal digit, then the base is 8; if `s[0]` is `'0'` and `s[1]` is either `'x'` or `'X'`, then the base is 16.

strtod expects *s* to contain a string of the form

```
[whitespace][+|-][digits][.digits][e|E][+|-][digits]
```

strtol expects *s* to contain a string of the form

```
[whitespace][+|-][0][x|X][digits]
```

RETURNS: *strtod* and *strtol* return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value would cause underflow or overflow, plus or minus `HUGE_VAL` is returned from *strtod* and `LONG_MAX` or `LONG_MIN` is returned from *strtol*, (according to the sign of the value), and errno is set to `ERANGE`.

EXAMPLE:

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

double getIntval(void) {
  char buffer[80];

  fgets(buffer, sizeof(buffer), stdin);
  return strto(buffer, NULL, 0);
  }
```

strtok

```
char *strtok(str, dlm)
char *str, *dlm;
```

strtok decomposes *str* into a sequence of tokens delimited by one or more of the delimiter characters of *dlm*. The tokens are returned via a series of calls to *strtok.*

The first call to *strtok* specifies *str,* while subsequent calls subtitute `NULL` for *str.*

Each call skips initial delimiters (i.e., `strspn(str,dlm)`), and then scans for the trailing delimiter (i.e., `strpbrk(str,dlm)`).

RETURNS: *strtok* returns a pointer to the first/next token in *str,* or `NULL` if there are no more tokens.

All tokens are terminated with `'\0'`.

SEE ALSO: `strcspn(), strspn()`

EXAMPLE:

```
/* break string into tokens /*

char *strtok(str, dlm)
char *str, *dlm;{
   char *beg, *end, *strpbrk();
   static char *nxt;

   if((beg = (str == NULL) ? nxt : str) == NULL)
      return NULL;
   beg = beg + strspn(beg, dlm);
   if(*beg == '\0')
      return NULL;
   if((end = strpbrk(beg, dlm)) == NULL)
      nxt = NULL;
   else{
      *end++ = '\0';
      nxt = end;
      }
   return beg;
   }
```

strupr

```
int strupr(src)
char src[];
```

strupr converts any lower-case characters in *src* to upper-case.

RETURNS: *strupr* returns a pointer to the converted string, *src*.

SEE ALSO: `strlwr()`

EXAMPLE:

```
/*
** convert string to upper case
*/

char *strupr(src)
char *src;{
    char *ret = src;

    for(; *src; src++)
        *src = toupper(*src);
    return ret;
    }
```

system

```
#include <stdlib.h>

int system(char cmd[]);
```

system invokes the DOS COMMAND.COM to execute *cmd* .

system uses the environment variable COMSPEC to locate COMMAND.COM.

RETURNS: *system* returns 0 if *cmd* was successfully executed, -1 otherwise.

EXAMPLE:

```
#include <stdlib.h>

int system(char cmd[]){
   char path[65], arg[129];

   getenv("COMSPEC", path);
   strcpy(arg, "/c");
   strcat(arg, cmd);
   if(exec(path, arg))
      return -1;
   return 0;
   }
```

Intentionally blank

tan

```
#include <math.h>

double tan(x)
double x;
```

tan computes the tangent of its radian argument *x*. The meaningfulness of the result depends upon the magnitude of the argument.

RETURNS: *tan* returns the tangent, or a huge number and sets *errno* to **ERANGE** at its singular points.

SEE ALSO: `acos(), asin(), atan(), cos(), sin()`

times

```
void times(buf)
char buf[9];
```

times formats the string *buf* with the current time as "hh-mm-ss".

If hh, mm or ss are less than 10, they will be formated with a space (0x20) as their first character.

RETURNS: *times* returns no value.

SEE ALSO: ctime()

EXAMPLE:

```
/*
** sleep n seconds
*/

sleep(n)
int n;{
    char cur[9], ref[9];

    times(cur);
    while(1){
        times(ref);
        if(strcmp(cur, ref))
            if(n-- == 0)
                return;
            else
                strcpy(cur, ref);
        }
    }
```

tolower, toupper

```
char tolower(c)
char c;

char toupper(c)
char c;
```

tolower converts upper-case letters to lower-case. *toupper* performs the opposite conversion.

RETURNS: Both functions return the converted character, or *c* unchanged if it isn't the correct case.

NOTE: These are functions rather than the usual macro implementation.

EXAMPLE:

```
/*
** if c is upper case, return lower case
** else return c
*/

tolower(c){
char c;{

    if(c >= 'A' && c <= 'Z')
        c -= 'A' - 'a';
    return c;
    }


/*
** if c is lower case, return upper case
** else return c
*/

toupper(c){
char c;{

    if(c >= 'a' && c <= 'z')
        c += 'A' - 'a';
    return c;
    }
```

ungetc

```
#include <stdio.h>

int ungetc(ch, fp)
char ch;
FILE *fp;
```

ungetc pushes the character *ch* back onto the file *fp*. The next call to *getc* or *fgetc* will return *ch* .

Only one character can be pushed back onto *fp* between calls to *getc* or *fgetc* .

RETURNS: *ungetc* returns *ch*, or -1 if it can't push the character back.

NOTE: *fseek* clears all pushed characters.

EOF (-1) can't be pushed.

SEE ALSO: `getc(), getchar()`

EXAMPLE:

```
/* get an unsigned number from console */

#define val(ch) (isdigit(ch) ? ch - '0':\
                 10 + tolower(ch) - 'a')

long ctol(base)
int base;{
    long num = 0L;
    int d, ch;

    if(base < 0 || base > 36)
        return val;
    while(isalnum(ch = getc(stdin)) &&
            (d = val(ch)) < base)
        num = num * base + d;
    ungetc(ch, stdin);
    return num;
    }
```

unlink

```
int unlink(oldFile)
char *oldFile;
```

unlink deletes the file *oldFile*. Under DOS 2.0 and higher, *oldFile* may contain a path specification.

RETURNS: *unlink* returns 0 if successful, or -1 if *oldFile* doesn't exist, is open, or if an error is detected.

utoa

```
#include <stdlib.h>

char *utoa(unsigned v, char s[], int r);
```

utoa converts *v* into a null terminated string at *s* . *r* specifies the base of *v* ; it must be in the range 2 — 36.

If *r* is 10 and *v* is negative, the first character of *s* will be the minus sign, '-'.

RETURNS: *utoa* returns a pointer to *s* .

NOTE: *utoa* is implemented as a macro

EXAMPLE:

```
#include <stdlib.h>

/* convert unsigned to string */

char *utoa(unsigned val, char *str, int rad) {
  return ltoa((long)val, str, rad);
  }
```

write

```
int write(handle, buf, count)
int handle;
char *buf;
unsigned count;
```

write writes *count* bytes from *buf* to the file *fp*.

write starts writing at the current position of *fp*. After the write, the current position has advanced *count* bytes.

RETURNS: *write* returns the number of bytes actually written, or -1 if an error occurred.

SEE ALSO: `fputc(), fputs(), printf()`

EXAMPLE:

```
/*
** Small Case block transfer
*/

xfer(ih, oh)
int ih, oh;{
   char *buf, *_memory();
   unsigned size, amt;

   freeall(256); /* min stack */
   buf = _memory() + 1; /* point to size */
   size = (unsigned *)buf & 0xF800; /* % 2K */
   buf += 2; /* point to buffer */
   do {
      amt = read(ih, buf, size);
      if(amt && write(oh, buf, amt) != amt){
         puts("xfer: write error);
         exit(2);
         }
      } while(amt == size);
   close(oh);
   }
```

Intentionally blank

_doint

```
extern unsigned  _rax, _rbx, _rcx, _rdx,
                 _rsi, _rdi, _res, _rds;
extern char   _carryf, _zerof;

void _doint(inum)
char inum;
```

_doint will cause software interrupt *inum* and may be used to call whatever routines are available in the particular machine.

_rax - *_rds* contain the values of the corresponding 8088 internal registers that are loaded and saved by *_doint* .

_carryf is the carry flag; *_zerof* is the zero flag

If *_rds* is set to -1, the current value of the DS register is used.

RETURNS: *_doint* returns no value. The interrupt may return values in *_rax*, ...

SEE ALSO: `_os()`

EXAMPLE:

```
/*
** get current cursor location via int 10H
**
/*

#define scr_row() (scr_curloc() >> 8)
#define scr_col() (scr_curloc() & 0xFF)

scr_curloc(){
  extern unsigned _rax, _rbx, _rdx;

  _rax = 0x0300; /* AH = 3 */
  _rbx = 0;
  _doint(0x10);
  return _rdx;
  }
```

_gets

```
int _gets(buf, max);
char buf[];
int max;
```

_gets obtains a string of not more than *max* - 1 characters from the console into *buf*.

Editing proceeds as with *gets* .

RETURNS: *_gets* returns the number of characters obtained, or 0 on end of file or an error.

SEE ALSO: `fscanf(), fread()`

NOTE: *_gets* doesn't return the CR character.

_in, _out

```
char _inb(port)
unsigned port;

unsigned _inw(port)
unsigned port;

void _outb(ch, port)
char ch;
unsigned port;

void _outw(wd, port)
unsigned wd, port;
```

_inb and *_inw* read the byte *ch* and word *wd*, respectively, from the indicated *port*.

_outb and *_outw* write the byte *ch* and word *wd*, respectively, of data out to the indicated *port*.

RETURNS: *_inb* and *inw_* return the byte or word read. There are no error values or codes.

EXAMPLE:

```
/*
** read comm port
*/

#define MCR (port + 4)
#define LSR (port + 5)
#define MSR (port + 6)

#define DSR 0x20
#define RDY 0x01

agetc(port)
int port;{
    _outb(1, MCR); /* set DTR */
    while(!(_inb(MSR) & DSR))
        ;   /* wait for data set ready */
    while(!(_inb(LSR) & RDY))
        ;   /* wait for data */
    return _inb(port); /* read data */
    }
```

_lmove
(small case model)

```
void _lmove(num, sp, sseg, tp, tseg)
char *sp, *tp;
unsigned num, sseg, tseg;
```

_lmove moves *num* bytes from the 8088 physical address at *sseg:sp* to *tseg:tp*. For example, to move the color display frame buffer at address `0xB800:0` to a local buffer (*_showds* provides the C program data segment — DS)

```
   _lmove(4000, 0, 0xB800, buffer, _showds());
```

RETURNS: *_lmove* returns no value.

SEE ALSO: `_move()`

NOTE: *_lmove* takes advantage of the 8088 instructions for a fast data move. It handles overlapping moves correctly so that

```
   _lmove(3920, 0, 0xB800, 80, 0xB800);
```

will move `0xB800:3919` to `0xB800:3999`, `0xB800:3918` to `0xB800:3998` etc. rather than propagating `0xB800:0`.

_memory
(small case model)

```
char *_memory();
```

RETURNS: *_memory* returns a pointer to the first free byte beyond the uninitialized data area in the small case model.

See the **Memory Management** discussion of the memory allocation area.

SEE ALSO: `malloc()`

EXAMPLE:

```
/*
** get the size of the malloc area
*/

struct {
    char stat;
    unsigned size;
    char data[1];
    };

maxMem(stack)
int stack;{  /* size of stack expansion area */
    char *mp, *_memory();

    freeall(stack);
    mp = _memory();
    return mp->size;
    }
```

_move

```
void _move(number, sourcePtr, targetPtr);
unsigned number;
char *sourcePtr, *targetPtr;
```

_move moves *number* bytes from *sourcePtr* to *targetPtr* .

RETURNS: *_move* returns no value.

SEE ALSO: `_lmove()`

NOTE: *_move* takes advantage of the 8088 instructions for a fast data move. It handles overlapping moves correctly so that

```
char buffer[80];

_move(79, buffer, &buffer[1]);
```

will move `buffer[78]` to `buffer[79]`, `buffer[77]` to `buffer[78]` etc. rather than propagate `buffer[0]`. Use *_setmem* to fill a range of memory with a value.

_OS

```
char _os(inum, arg);
char inum;
unsigned arg;
```

_os provides an elementary interface to the BIOS.

inum goes into AH, *arg* into DX, and an `int 21H` is executed.

RETURNS: *_os* returns the value returned from the interrupt in the 808X AL register.

SEE ALSO: `_doint()`

EXAMPLE:

```
/*
** use DOS function 09H to print a string
*/

main(){
   _os(9, "Hello World!!$")
   }
```

_peek, _poke

```
char _peek(sp, sseg);
char *sp;
unsigned sseg;

void _poke(ch, tp, tseg);
char ch, *tp;
unsigned tseg;
```

_peek is used to retrieve a byte *ch* from the 8088 physical address at *sseg:sp* .

_poke is used to store the byte *ch* of data to the 8088 physical address at *tseg:tp* .

RETURNS: *_peek* returns the byte, *_poke* returns no value.

EXAMPLE:

```
/*
** get environment strings - small case
** assumes environ is lower in memory than PSP
*/

extern unsigned _pcb; /* PSP address */

getEStr(){
    unsigned  _memory(), env, size;

    env = _peek(0x2D,_pcb); /*high-order byte*/
    env = (env << 8) | _peek(0x2C,_pcb);

    size = (_pcb - env) << 16;

    _lmove(size, 0, env, _memory(), _showds());

    return _memory();
    }
```

_setmem

```
void _setmem(dst, number, ch);
char *dst, ch;
unsigned number;
```

_setmem sets *number* bytes of memory starting at *dst* to the byte value *ch* .

RETURNS: *_setmem* returns no value.

SEE ALSO: `strset(), _move()`

EXAMPLE:

```
/*
** zero an array - use instead of
**  for(i=0; i<SIZE; i++) data[i] = 0;
*/

#define zArray(a) _setmem(a, sizeof(a), 0)


double test[1024];

zArray(test);
```

_setsp

```
void _setsp(sp)
char *sp;
```

_setsp sets the stack pointer (the SP register) to *sp* .

RETURNS: *_setsp* returns no value.

NOTE: In small case, *sp* can range from 0 to 0xFFFF. In large case, the range is 0 to the size of the stack - 1.

_showcs, _showds, _showsp

```
unsigned _showcs();
unsigned _showds();
unsigned _showsp();
```

RETURNS: *_showcs* returns the paragraph address of the code segment (the CS register).

_showds returns the paragraph address of both the data and stack segment (the DS and SS registers) in small case and the data segment in large case.

_showsp returns the contents of the SP register in small case and SS:SP in large case.

Appendix A

Messages

ASM88 Assembler Messages

Banner and Termination Messages	A.1
ASM88 Fatal Error Messages	A.2
ASM88 Error Messages	A.2

BIND Messages

Banner and Termination Messages	A.7
BIND Fatal Error Messages	A.7
BIND Warning Messages	A.8

C88 Compiler Messages

Banner and Termination Messages	A.9
C88 Fatal Error Messages	A.10
C88 Error Messages	A.12
C88 Warning Messages	A.18
ASM88 Messages from C88	A.18-1

CLIST Messages

Banner and Termination Messages	A.18-1
CLIST Fatal Error Messages	A.18-1

D88 Messages	A.19

LIB88 Messages

Banner and Termination Messages	A.21
LIB88 Fatal Error Messages	A.21
LIB88 Warning Messages	A.22

SEE Messages

Banner and Termination Messages	A.23
SEE Error and Status Messages	A.23

ASM88 Messages

Banner and Termination Messages

```
ASM88 8088 Assembler V1.5 (c) Mark DeSmet, 1982-86
```

(various error messages)

```
end of ASM88  0016 code  0000 data  1% utilization
```

The 'code' number is in hex and tells how many bytes of code were produced. The 'data' number is similar and tells how many bytes of data were produced. The utilization percentage shows how full the symbol table was.

Sample of list output:

```
ASM88 Assembler   BLIP.A
                   1  ;TOUPPER.A convert a charcter to upper case
                   2
                   3  CSEG
                   4  PUBLIC TOUPPER
                   5
                   6  ; character = toupper(character)
                   7
0000 5A            8  TOUPPER:   POP   DX            ;RETURN ADDRESS
0001 58            9             POP   AX            ;CHARACTER
0002 3C61         10             CMP   AL,'a'        ;IF LOWER THAN 'a'
                  11             JC    TO_DONE       ;DO NOTHING
0004 3C7B         12             CMP   AL,'z'        ;OR IF ABOVE 'z'
                  13             JNC   TO_DONE       ;DO NOTHING
0006 2C20         14             SUB   AL,'a'-'A'    ;ELSE ADJUST
0008 B400         15  TO_DONE:   MOV   AH,0          ;RETURN AN INT
000A FFE2         16             JMP   DX            ;RETURN
```

ASM88 prints two categories of messages: fatal errors and errors. As with C88, the fatal errors are caused by I/O errors or similar. Errors are simply syntax errors in using the language. When a fatal error is detected, ASM88 prints a message and stops. An error does not stop the assembler, but it stops writing the object module to run faster. If errors are detected, the object module is never good.

ASM88 Fatal Errors

cannot close <file> — the file could not be closed. An I/O error occurred.

cannot create <file> — the named file could not be created. The name is a temporary name or the name of the object or list file. This message usually means the drive is full (see 'T' option).

cannot open <file> — the named source or include file could not be found.

cannot read <file> — the named file could not be read. Usually means an I/O error was detected.

cannot unlink <file> — the temporary file could not be deleted. An I/O error occurred.

cannot write <file> — the named file could not be written. An I/O error was detected. Usually means the disk drive is out of space.

internal error in jump optimization — the assembler became confused optimizing branches.

no input file — no filename followed the ASM88 when invoked.

too many labels — only 1000 labels are allowed.

too many symbols — the assembler ran out of symbol space. The source program should be broken into smaller modules.

ASM88 Error Messages

Error messages have the form:

```
44  mov  #44,a3
error: illegal mnemonic
```

or, if the error was found in an include file:

```
44  mov  #44,a3
file:2:SCREEN.A  error: illegal mnemonic
```

The messages are:

address must be in DSEG — address constants can only be in DSEG because constants in CSEG are not fixed up at run time.

bad DS value — a constant expression must follow the DS.

bad include — the correct form for an include statement is:

```
include "filename"
```

bad LINE value — the line statement should be followed by a constant.

cannot label PUBLIC — a 'public' statement cannot have a label.

data offset must be an unsigned — an attempt was made to use an offset in a byte or long constant.

DS must have label — storage cannot be reserved without a name.

DS must be in DSEG — storage can only be reserved in DSEG.

duplicate label — the label on the line was defined previously.

equate too deep — an 'equ' may reference a prior one, but only to a depth of four.

illegal expression — the expression had an illegal operator or is somehow invalid.

illegal operand — an operand had a type that was not legal in that context.

illegal reserved word — a reserved word was found in the wrong context.

illegal ST value — the index to a floating point stack element must be in the range 0 to 7.

incorrect type — only 'byte', 'word', 'dword', and 'tbyte', are allowed following the colon to type a public.

impossible arithmetic — an arithmetic operation has operands incompatible with the 8086 architecture, for example:

```
        add word [bx], word[si]
```

in wrong segment — a variable or label is being defined in a segment other than the segment of its 'public' statement. Remember that 'public' statements must be in the correct segment, following 'dseg' or 'cseg' as appropriate.

invalid BYTE constant — a byte constant was needed, but something else was found.

invalid constant — the instruction needed a constant and something else was found.

invalid DD constant — the value of a 'DD' must be a constant expression.

invalid DW constant — the value of a 'DW' must be a constant expression or a variable name. In the latter case, offset is assumed. The statement:

```
        dw   offset  zip
```

is illegal since offset is already implied. Just use:

```
        dw   zip
```

invalid offset — an offset of the expression cannot be taken.

line too long — the maximum input line to ASM88 is 110 characters.

mismatched types — the types of the two operands must agree.

example:

```
        db   chr
        add  ax,bl          ;illegal
        add  chr,ax         ;illegal
        add  word chr,ax    ;legal
```

misplaced reserved word — a reserved word was found in an expression.

missing : — the '?' operator was missing the colon part.

missing) — mismatched parentheses.

missing] — mismatched braces in an address expression.

missing ':' — labels to instructions must be followed by a colon. This message also prints when a mnemonic is misspelled. The assembler thinks that the bad mnemonic is a label without a colon.

missing EQU name — an equate statement lacks a name.

missing type — the memory reference needs a type. In the case of 'public's defined elsewhere, the type can be supplied by ':byte' or ':word' on the public statement. In the case of anonymous references, the 'byte' or 'word' keyword must be used, for example:

```
public    a:byte
inc  a              ; illegal
inc  byte a         ; legal
inc  es:[bx]        ; illegal
inc  es:word[bx]    ; legal
```

need constant — something other than a constant expression followed a 'ret'.

need label — a jump relative was made to something other than a label. 'jmp's may be indirect but 'jz's etc. can only jump to a label.

nested include — an included file may not include another.

not a label — only names can be public.

RB must have label — an 'RB' statement must have a label.

RB must be in DS — 'RB's must follow a DSEG directive as they can only be in the data segment. 'DB's can be in the code segment.

RW must be in DS — as above.

too many arguments — the instruction had more operands than allowed or the last operand contains an illegal op-code.

undefined variable <name> — the name is referred to but not defined or listed as public.

unknown mnemonic — the mnemonic is illegal.

BIND Messages

Banner and Termination Messages

```
Binder for C88 and ASM88     V2.0     (c) Mark DeSmet, 1982-87
end of BIND         9% utilization
```

BIND Fatal Error Messages

BIND prints the message, prints 'BIND abandoned' and quits.

bad argument — an argument is illegal.

bad object file<name> — the object or library file contains an illegal record.

bad stack option — the 'S' option should be followed by one to four hex digits.

cannot close <file> — I/O error occurred.

cannot create <file> — I/O error or disk out of room. On MS-DOS 2.0 and later, make sure that the CONFIG.SYS file contains a FILES=20 command.

cannot open <file> — the object file could not be found. On MS-DOS 2.0 and later, make sure that the CONFIG.SYS file contains a FILES=20 command.

cannot read <file> — I/O error occurred.

cannot seek <file> — I/O error occurred.

cannot write <file> — I/O error or disk out of room.

different segments for - <name> — the public is declared in different segments in different modules — probably both as a function and as a variable.

illegal overlay number — in the overlay options -Vnn and -Mnn, the value nn must be between 1 and 39 in ascending consecutive order.

multiply defined <name> — the same public appears in two modules.

over 100 arguments — BIND only allows 100 arguments, including arguments in -F files.

over 64K code — a Small Case program has over 64K of code. See the description of BIND overlay support.

over 64K data — a Small Case program has over 64K of data. This is not supported. You will have to move some data to locals or use overlays.

over 300 modules — only 300 modules can be linked together. The supplied library only contains about 60 modules.

too many filenames — there are only 2000 bytes reserved for all filenames.

too many labels in <name> — a module in the named file had over 1000 labels.

too many total PUBLICS in <name> — symbol table has overflowed. The named file was being read when the overflow occurred.

BIND Warning Messages

undefined PUBLIC - <name> — the name is referenced, but not defined in any module. BIND will complete and the resulting .EXE module may execute as long as the undefined PUBLICs are not referenced. If they are referenced, then the result is undefined.

C88 Messages

Banner and Termination Messages

```
>C88 Compiler   V3.1   Copyright Mark DeSmet 1982-1988
end of C68    001A code    0012 data     1% utilization
```

OR

```
>C88 Compiler   V3.1   Copyright Mark DeSmet 1982-1988
```
(various error messages)

```
    Number of Warnings = 2     Number of Errors = 5
```

The first form of termination means the compilation was successful. The 'code' number is in hex and tells how many bytes of code were produced. The 'data' number is similar and tells how many bytes of data were produced. The utilization percentage is the worst case of a number of compiler limits. If it nears 100% it usually means that the largest procedure should be broken into smaller procedures.

The second form means the compilation failed. Error messages are explained in the following section. If any errors were detected, the compiler will stop trying to generate code and will stop as soon as all the source has been read. This syntax check' mode is fast and allows the programmer to correct the program with a minimum of delay. If only warnings are detected, but no errors, the compilation will end normally and produce a .O file.

C88 produces four categories of messages: fatal errors, errors, warnings and errors detected by the assembler. Fatal errors are usually caused by I/O errors but compiler errors are also in this category. When a fatal error is detected, the compiler will print a message and quit. Errors are caused by syntax errors. If C88 is invoked from SEE, it returns to SEE upon the first error, otherwise it reports all such errors and then quits. Warnings are produced by correctable errors and the compiler continues. Since the compiler only uses ASM88 as pass 3 if the **-a** option or the #asm option is used, assembler detected errors are possible but rare. When they occur, the object module will not be usable.

It is easy to tell the category of an error. After a fatal error, the compiler stops without printing a termination message. Errors and warnings have a distinctive format which includes the word 'error' or 'warning'. Assembler errors print the assembler line that was found offensive.

C88 Fatal Error Messages

The pass 2 fatal errors like 'bad expression' are compiler errors, but the error is usually caused by missing the problem in pass 1 and printing a reasonable message. If you get one of these errors, please send your program to C Ware, but you can probably find and eliminate the statement that caused the problem. Don't be frightened by seeing these errors listed; you will probably never see any of them.

bad expression — this indicates a compiler error. Printed by pass 2.

bad GOTO target — attempt to goto something other than a label.

break/case/continue/default not in switch — a case or default statement must be within a switch. A break statement must be in a while, do...while, for, or switch. A continue statement must be in a while, do...while, or for statement.

cannot address — illegal use of '&' operator. Printed in pass 2.

cannot close <file> — the file could not be closed. An I/O error occurred.

cannot create <file> — the named file could not be created. The name is a temporary name or the name of the object or assembler file. This message usually means the drive is full (see 'T' option).

cannot open <file> — the named source or include file could not be found.

cannot read <file> — the named file could not be read. Usually means an I/O error was detected.

cannot unlink <file> — the temporary could not be deleted. An I/O error occurred.

cannot write <file> — the named file could not be written. An I/O error was detected. Usually means the disk drive is out of space.

error in register allocation — compiler error in pass 2.

divide by zero — a constant expression evaluated to a divide by zero. Should never happen.

E option not valid from SEE — You have specified the E option on the C88 command line from SEE. Either remove the option, or exit SEE and run C88 from the command line prompt.

function too big — a function is too big for the compiler. The 'Utilization' number reflects this limit so there is normally plenty of warning. The solution is to break large procedures into smaller ones.

illegal initialization for <name> — only constant expressions and addresses plus or minus constant expressions can be used in initialization and the initialization must make sense. For example

```
        int a=b+2;
```

this error is fatal because it is not discovered until pass 2.

no cases — a switch must have at least one case.

no input file — You must specify the name of the source file.

out of memory — the compiler ran out of symbol space. The 'utilization' numbers warn when a program is about to exceed this or any other compiler limit. The compiler can use up to 100K, so adding memory may be a solution. If not, the only remedy is the painful surgery required to reduce the total number of externals and locals defined at one time.

pushed — compiler error in pass 2 code generation. It can be eliminated by simplifying the expression.

stdin not a device — you have specified '-' as the filename, but `stdin` is not a file (`isatty()` is true). You must either redirect `stdin`, or use a pipe.

stuck <register> — same as 'pushed'.

too many cases — currently, a switch statement can only contain 128 case statements.

too many externals — the compiler currently has a limit of 500 static's or extern's.

too many fors/too many whiles — whiles, do-whiles, switches and for statements can only be nested 10 deep.

C88 Error Messages

Errors are printed with the following format:

```
23 if (i < 99 $$ {
     error:Need ()
```

Or, if the error was detected in an include file:

```
23 if (i < 99 $$ {
    file:<include file> error:Need ()
```

The number preceding the source line is the line number. To find the line , edit the file and issue the command 'nnnJ' where nnn is the number of the reported line.

The '$$' shows how far into the line the compiler was before the error was detected. For example, the '$$' will print immediately BEFORE an undefined variable.

If you get a lot of errors on a compile, don't panic. A trivial error probably caused the compiler to become confused. Correct the first few errors and re-compile.

## can't be first — the macro concatenation operator must occur between tokens.

#(#) can't be last — macro text must follow both the concatenation and stringify operators.

#asm option not on — a `#asm` directive is found without the extended keyword switch on. Use the command line option **px**, or `#pragma ex`

#undef identifier not defined — the identifier has not been `#define`'d.

bad control — the directive following the # is unknown.

bad declaration — the declaration of a variable was illegal.

bad include — the #include must be followed by "name" or <name>, or a macro that evaluates into one of the previous two forms.

bad label — a colon is not preceded by a label name.

bad member declare — the declaration of a member is illegal.

bad member storage — an attempt was made to declare a member static or external. Members have the storage type of their struct or union.

bad parameter declare — an illegal declaration of an argument or the name of the argument was spelled differently in the procedure heading and in the declaration.

bad statement — illegal statement.

bad STRUCT declare — an error has occurred in a `struct` declaration.

cannot initialize extern — `extern` variables are defined, and possibly initialized, elsewhere. You can have both an `extern` declaration and definition of a variable in the same source file.

cannot #undef predefined macros — the predefined macro names (`__FILE__`, `__LINE__`, ...) cannot be undefined.

cannot redefine predefined macros — the predefined macro names cannot be redefined.

case range option not on — a `case` *cexpr* .. *cexpr* was found and the extended keywords option is off. Use the **px** command line option or `#pragma ex`.

defines too deep — #define may reference another, but there is a limit. When #defines are expanded, the number of active #defines cannot exceed 32.

duplicate argument — an attempt was made to declare an argument twice.

duplicate enum — `enum`'s names must be unique.

duplicate label — two labels have the same name.

EOF within comment beginning at line *nnnn* — end of file was found inside a comment which began at line *nnnn* . A '*/' is missing.

EOF in macro argument — end-of-file was found while evaluating a macro argument. An unterminated comment or string is the most likely reason.

field needs constant — the size of a bit field must be a constant expression with a value of 1 to 16.

illegal address — attempt to use the '&' (take address of) operator on something that is not an lvalue. '&44' will generate this error. An address can only be taken of a variable, procedure, string or label.

illegal arithmetic — the requested pointer arithmetic doesn't make sense.

illegal assignment — only a pointer, long, or constant can be assigned to a Large Case pointer. Note: this is a pass 2 error — the **-c** (checkout option) must be used to get the line number of the error.

illegal define — a #define has unmatched parentheses or the #define parameters are illegally specified.

illegal double constant — a `double` or a `float` was specified using hexadecimal notation and the result is not 8 or 4 bytes long.

illegal external declaration — caused both by an illegal data or procedure declaration and improperly nested braces. If the line is supposed to be part of a procedure (e.g. i=0;), the latter is the case. Verify that every '{' outside of a comment or quoted string has a matching '}'. Note: a prior error may have caused the compiler to lose track of a '{'.

illegal index — a pointer cannot be used as an array index

illegal indirection — something other than a pointer has been used as a pointer.

include nesting too deep — includes can only be nested 20 deep

illegal structure assignment — the two operands of an assignment operator are not the same structure, or the same size.

illegal type — an invalid type specifier combination has been found.

illegal use of FLOAT — floating point numbers cannot be used as pointers.

invalid digit-sequence in #line — the symbol following the `#line` directive doesn't evaluate to a number.

invalid identifier in #ifdef/#ifndef — the symbol following the `#ifdef/ #ifndef` is not a vaild name (doesn't begin with a letter or underscore).

invalid identifier in #undef — the symbol following the `#undef` is not a vaild name.

invalid identifier in defined() operator — the symbol following the `defined` operator is not a vaild name.

invalid parameter — a parameter of a macro is not a valid name.

invalid string-literal in #line — the symbol following the digit-sequence in the #line directive doesn't evaluate to a string literal.

line must be constant — a #line control must be followed by a decimal constant.

line too long — the maximum line length is 509 bytes.

macro buffer overflow — more than 1024 bytes of argument text, or more than 32 arguments were found.

member not in structure — the variable following a '.' or '->' operator is not a member of the `struct` or `union` that preceeded the operator.

missing ";", "(", ")", "[", "]", "{", "}", ":", "|" — the indicated "" character is needed at this point. A multitude of errors can cause these messages. The error might be fixed by inserting the indicated character where the '$$' prints, but the item following the '$$' could be illegal.

missing ' — a character constant (e.g. 'A','01') can only contain one or two characters.

missing argument — fewer arguments are supplied in a function call than were specified in the function prototype, or the argument list of a call had two adjacent commas.

missing arguments — a #define was defined with arguments but used without arguments.

missing dimension — an array dimension was missing in an expression or statement. Either int x[][]; or x[]=1;.

missing end of #asm — an #asm block was not ended with a #.

missing expression — an expression is needed here. An example of a missing expression is i=;.

missing operand — an operator without an operand was found. An example of a missing operand is ++;

missing while — a 'do ... while' is missing the ending 'while'.

must have constant — C syntax requires a constant value at this point.

must return float — in a function declared as returning a `double` or a `float`, the last statement is not a `return` *floating-type* .

must return structure — in a function declared as returning a structure *structure-type* either the last statement is not a `return` *structure-type* or a `return` of something other than *structure-type* is found.

need () — the expression following an 'if' or 'switch' or 'while' was not surrounded by parentheses.

need '{' for STRUCT initialization — the initial values used to initialize a structure must be surrounded by braces.

need closing parenthesis — a macro parameter definition does not end in ')'.

need constant — a 'case' prefix must be followed by an integer constant expression.

need label — a goto must reference a label.

need lval — an lvalue is needed here. An lvalue is, roughly, something that can be changed with an assignment. The statement: 2=4; will produce this error.

need member — the '.' or '->' operators were followed by something other than a member name.

need structure — in *id . member* or *id -> member* *id* is not a `struct` or `union`, or a pointer to one.

not enough #include buffer space — DOS cannot allocate space for an `#include` file buffer. Reduce RAM-disk size or remove TSR's to increase available RAM.

not enough arguments — fewer arguments are supplied in a function call than were specified in the function prototype.

only STATIC and EXTERN allowed at this level — an attempt was made to declare an 'auto' outside of a procedure.

parameter must follow # — the macro stringify operator must be followed by a macro parameter.

return lacks argument — if a function is declared as returning a value then "return;" is illegal. Use "return 0;" if the value is unimportant.

sizeof operator not allowed in #if/#elif — sorry, this is an ANSI requirement.

sorry, must have dimension for locals — the compiler does not accept char a[]={1,2,3}; and similar for auto variables. Declare the variable static or include an explicit dimension.

sorry, no string initialization of AUTO — the compiler cannot accept char a[]="abc"; and similar for auto variables. Declare the variable static if possible, otherwise use _move.

string too long — a string cannot exceed 255 characters. Usually means that a '"' is missing. Use the string concatenation feature to create long strings.

too many arguments — more arguments are specified in a function call than were specified in the function prototype.

undefined structure — a structure is referenced without being defined.

undefined variable — an unknown id was found as an argument to a function call.

unknown control — the word following a '#' is not a control word. '#while' would cause this error.

unmatched " — either the end of line or end of file was found in a string. This usually means that a " is missing. If your string is too long for one line, continue with a \ (backslash) at the end of a line and continue in column one of the next. If you want a new line in a string, use \n.

wrong number of arguments — a macro was used was used with the wrong number of arguments.

C88 Warning Messages

Warnings indicate a change in syntax (as in the case of structures), or suspicious code that is probably OK.

argument type conversion — a function argument doesn't agree with the type of the corresponding argument prototype. The argument is cast to the prototype. This warning is usually supressed. Use the **pw** command line option or `#pragma w`

conflicting types — an external or static was declared twice with different types. Usually caused by an include file declaring a variable incorrectly or by a program such as:

```
main() {
     char ch;

     ch=zipit();
     }
char zipit(ch)
char ch; {

     return ch;
     }
```

the call of zipit implicitly declares it to be a function returning an integer. The line 'char zipit(ch)' would be flagged as an error. The fix is to include:

```
char zipit();
```

above the call so the function is declared correctly before use.

member not in structure — the member identified by struct.member or by ptr->member is not a member of the specified structure. A `(void *)` pointer will select any member of an anonymous structure.

must return float — a float or double function must end with a return statement that returns a value.

Note: The following functions ends with an `if` statement

```
double x(){if (1) return 1.;}.
```

returns structure — the current function has been declared as returning a structure. This is to warn you that the entire structure, and not a pointer to it, is being returned. This warning is usually supressed. Use the **pw** command line option or `#pragma w`

structure assignment — the structure named as a parameter will be pushed on the stack rather than a pointer to the structure, as was the case in previous releases. This warning is usually supressed. Use the **pw** command line option or `#pragma w`

undefined variable — the variable has not been defined. It is assumed to be an auto int.

C88 ASM88 Errors

In theory, any ASM88 error message can be produced by a C88 compile gone bonkers but I have only seen the 'cannot write <name>' errors caused by lack of disk space.

CLIST Messages

Banner and Termination Messages

```
CLIST   V1.3  (c) Mark DeSmet, 1982,83,84
end of CLIST
```

CLIST Fatal ErrorMessages

All messages indicate fatal errors. CLIST prints the message, prints 'CLIST abandoned' and quits.

cannot close <file> — I/O error occurred.

cannot creat <file> — I/O error or disk out of room.

cannot open <file> — the source file could not be found.

cannot read <file> — I/O error occurred.

cannot write <file> — I/O error or disk out of room.

no input file — no list of files followed CLIST on the invocation line.

out of memory — CLIST ran out of room. Break the list of files in two.

D88 Messages

*** Control C *** — The user typed control-C or control-break. If control-C is typed while a user program is executing, the program cannot be restarted.

cannot open <filename> — Cannot open the named file for the List of Quit-Initialize command.

cannot open xyzzy — The module containing the `main()` function was not compiled with the **-c** switch.

cannot read <filename> — The named file could not be read. Probably an I/O error.

cannot repeat — Again can only follow Again, Display, List or Unassemble commands.

illegal address — The & operator was applied to something not in memory, e.g. &1.

illegal assignment — An attempt to assign an expression to a constant was made. Only memory references and register can be changed.

illegal command — The command letter is not vaild.

illegal operand — This is a catch-all error; it just means that the expression could not be parsed correctly.

illegal value — The break numbers are 1, 2, or 3.

invalid symbol — The name is not in the symbol table. Probably a typo or missing 0 before a hex constant.

line not found — The line is unknown. Only executable lines have number records. Other lines cannot be referenced by number. The file may not have been compiled with the -C option.

missing) missing] missing " missing ' — Unmatched bracketing character.

need a number — A line number contained something other than a digit. No expressions are allowed.

normal end — The program being debugged executed an `exit()` call.

not in a C procedure — The Proc-step command can only be executed when the debugger knows which procedure is being debugged. The Step command can be used.

some symbols lost — The .CHK is greater than 55K bytes. Recompile those modules you don't wish to debug without the **-c** switch and rebind to reduce the size of the .CHK file.

LIB88 Messages

Banner and Termination Messages:

```
Librarian for C88 and ASM88   V2.1    (c) Mark DeSmet 1982,83,84,85
-TOUPPER_
-ISDIGIT_
-ISALPHA_        ISUPPER_         ISLOWER_          ISSPACE_
ISALNUM_         ISASCII_         ISCNTRL_          ISPRINT_
ISPUNCT_
-TOLOWER_
end of LIB88        12% utilization
```

The list of code publics is only printed if the -P option is employed. A minus sign in column one indicates the start of a new module.

LIB88 Fatal Error Messages

LIB88 prints the message, prints 'LIB88 abandoned' and quits.

bad argument <argument> — the option is illegal.

bad object file<name> — the object or library file contains an illegal record.

cannot close <file> — I/O error occurred.

cannot creat <file> — I/O error or disk out of room.

cannot open <file> — the object file could not be found.

cannot read <file> — I/O error occurred.

cannot write <file> — I/O error or disk out of room.

no input file — no list of files followed LIB88 on the invocation line.

over 100 arguments — LIB88 only allows 100 arguments, including arguments in -F files.

over 300 modules — only 300 modules can be linked together. The supplied library only contains about 60 modules.

too many dependencies in <name> — there is a total of over 1500 dependencies between modules.

too many total PUBLICS in <name> — symbol table has overflowed. The named file was being read when the overflow occurred.

LIB88 Warning Messages

warning: circular dependencies — two modules reference each other; this is OK if the first is always included whenever the second one is. The -N (need) option will kill this message.

SEE Messages

Banner and Termination Messages

When the SEE editor reads in a file to edit, the menu line is replaced by the banner message:

```
SEE (TM): Screen Editor V3.0: Copyright 1982-1987  Michael Ouye
```

When the editor is exited, the message line prints the message:

```
    bye! <filename>
```

SEE Error and Status Messages

As commands are executed, the editor will display a number of status messages on the message line:

### characters — This message is displayed whenever a file is edited and when the Quit command is invoked. It shows the number of characters contained by the file.

bad command — This message is printed when there is no command that corresponds with the character typed.

bad tag name — This message is displayed when a letter besides A,B,C, or D was typed for a tag name.

can't find "<string>" — This message is displayed when a request to find the string fails.

can't write to file <filename> try again? (y/n) — An error occurred while attempting to write the file out to the disk. Type 'Y' to try to write the file to the same filename. Type 'N' to abort the attempt and use the Quit-Write command to write the buffer out to a different file.

hit a key to continue — This message is displayed during the List command to indicate that the next screenfull of text should be displayed.

ignore the changes? (y/n) — This message is printed when the memory buffer has been modified and not saved to disk and the buffer is about to be reinitialized with the Quit-Initialize command or the editor is about to be exited with the Quit-Exit command. Type 'Y to continue the command, or 'N' to abort the command.

no input file — This message is printed when the Update or BAKup commands are executed but no file was specified on the command line. Use the Quit-Write command to write the buffer out to the disk.

reading file: <filename> ... — This message appears whenever a file is read into memory. The completion status of the read operation is appended to the end of the message. If everything goes well, the word "completed" will be appended to the end of the message. Otherwise, the editor will append the string "can't read file" if an error occurred while attempting to read the file.

recording Macro F#, use Macro key to finish recording — This message is continually displayed as long as a Macro is being recorded. To end the Macro and the message, reinvoke the Macro command by typing the letter 'M'.

Appendix B

The ASM88 Assembly Language

Identifiers	B.1
Constants	B.1
Expressions	B.2
Registers	
General Registers	B.2
Byte Registers	B.3
Segment Registers	B.3
Addressing Modes	B.3
8086 Flags	B.4
Address Expressions	B.5
Address Typing	B.5
Comments	B.6
Assembler Direectives	B.6
Reserving Storage	B.8
Differences between ASM86 and ASM88	B.8
8086 Instructions	
Elements of Instructions	B.9
Instructions	B.9
8087	B.23
Control Word	B.24
Status Word	B.25
Tag Word	B.25
Condition Codes	B.26
8087 Instructions	B.27

Identifiers

Identifiers must start with a letter (A-Z, a-z, _), may contain digits, and have a maximum length of 31 characters. Upper and lower case letters are distinct so ABC, abc and Abc are three distinct identifiers.

Constants

Constants are binary, octal, decimal, hex, floating point, or string.

Binary constant:	`ddddb` or `ddddB` where *d* is 0 or 1.
Octal constant:	`ddddo` or `ddddO` or `ddddq` or `ddddQ` where *d* is between 0 and 7.
Decimal constant:	`[-]ddddd` where *d* is between 0 and 9.
Hex constant:	`ddddh` or `ddddH` where *d* is 0 to 9, a to f, or A to F.
Floating constant:	`[-]ddd[.ddd][[+|-]Edd]` where *d* is between 0 and 9.
String constant:	`'dddd'` where *d* is `\n` or `\N` for LF, `\t` or `\T` for TAB, '' for the single quote, `\ooo` where the `ooo` must be octal digits and the result is the corresponding character, or any other character.

After a DD (define double-word) mnemonic, constants that contain a period or 'E' exponent are single precision floating point. Other constants are signed four byte integers. After a DQ (define quad-word) mnemonic, constants are double precision floating point. A string constant after DB may have up to 80 characters. In any other place, constant expressions are allowed and the result has a range of 0 to 65535. There is no warning on overflow.

Expressions

All expressions operate on unsigned 16 bit constants. There is no warning when overflow occurs. Caution: multiplying or dividing negative constants will not give the expected results. -3/-1 is not 3.

The operators are listed in order of precedence.

`&`	binary and.
`== !=`	equality test and inequality test. Result is 0 (false) or 1 (true).
`+ -`	plus and minus.
`* /`	multiply and divide.
`& offset + - ! ~`	(& and offset are the same). plus minus not exclusive-or.

Registers

The 8086 has eight fairly general purpose registers and four segment registers. All registers are 16 bits wide.

General Registers

The following registers can be used in arithmetic or whatever but all have some specialized use.

AX	Some instructions have shorter forms using AX so AX is usually heavily used as an accumulator. MUL and DIV require AX. IN and out use AX.
BX	Used for addressing or for general purposes.
CX	Used by LOOP and JCXZ. Also used to contain a shift count.
DX	Used by MUL and DIV. Also used for variable port IN and OUT.
SI	Used for addressing and string instructions.
DI	Used for addressing and string instructions.
BP	Used as a stack pointer to access locals and arguments. Caution: C programs require BP to be preserved across calls.
SP	The stack pointer. Used by CALL and RET. Be very careful when manipulating SP.

Byte Registers

Each byte in the first four general registers can be addressed separately.

AH is the high byte of AX, AL is the low byte. BX, CX, and DX are similar.
The byte registers are: AH, AL, BH, BL, CH, CL, DH and DL.

Segment Registers

DS	Points to the data segment. The initialization code makes DS address the data in DSEG. All memory references that are not relative to BP and that do not include an explicit segment register override, refer to the segment addressed by DS.
ES	Points to additional data segment. C only uses ES when doing a move. The string instructions (movsb,cmpsb etc.) implicitly reference ES:[DI] for the target. ES may be changed by any routine and is generally used to address data outside of DSEG and CSEG.
SS	Points to stack segment. C initialization sets SS to DS. This equivalence is important for C programs so that they can create pointers to arguments or locals which are on the stack. When it is necessary to change SS, a load of SP must immediately follow.
CS	The code segment. CS is set to CSEG by initialization.

Addressing Modes

Only certain registers can be used to reference memory. The following are the permissible combinations.

```
[BX+SI+displacement]
[BX+DI+displacement]
[BP+SI+displacement]
[BP+DI+displacement]
[SI+displacement]
[DI+displacement]
[BP+displacement]
[BX+displacement]
[displacement]
```

Names can be included in an address, e.g. `blap[BX]`. The offset of the name is simply added to the displacement.

Addresses that include BP are assumed to be SS relative. Other addresses are assumed to be in DSEG, addressed by DS. To override this assumption, prefix an address with 'DS:', 'ES:', 'SS:', or 'CS:'. The assembler automatically provides the prefix necessary for variables declared in CSEG.

Sample Addresses

```
hello   db   'Hello',0
save    dw   0
again:  mov  save,99          ;moves 99 to save
        mov  hello+3,'p'      ;changes 'Hello' to 'Helpo'
        mov  bx,4             ;sets bx to 4
        mov  hello[bx],'!'    ;changes 'Helpo' to 'Help!'
        mov  ax,offset again ;moves offset of again to ax
        mov  ,ax              ;moves ax to save
        mov  ax,0             ;sets ax to zero
        mov  es,ax            ;sets es to ax which is zero
        mov  ax,es:[bx+4]     ;moves word at 0:8 to ax.
                              ;offset of NMI interrupt.
```

8086 Flags

The flags are set 1) directly, 2) as side effects of arithmetic instructions, and 3) by POPF (pop flags) and IRET (interrupt return). If you do a PUSHF (push flags) followed by a POP, they will appear as a word with the following format:

```
----------------------------------------------------------------
| X | X | X | X | OF| DF| IF| TF| SF| ZF| X | AF| X | PF| X | CF|
----------------------------------------------------------------
```

CF	carry flag. Set by arithmetic instructions to indicate unsigned overflow. The carry flag is not set by INC and DEC. Can be set with STC and turned off with CLC.
PF	parity flag. Set by arithmetic instructions to indicate parity. On for zero parity which means an even number of bits are on in the result.
AF	auxiliary carry flag. Used in BCD arithmetic.
ZF	zero flag. Set to 1 or true if the result of arithmetic instruction is zero.
SF	- sign flag. Set by arithmetic instructions if the sign (highest) bit is on.
TF	trap flag. Set by debuggers to cause single stepping. Can only be set by IRET.

IF	interrupt enable flag. Set by STI, turned off by CLI and interrupt.
DF	direction flag. Determines direction of string instructions. Set off, which means increasing SI and DI, by CLD. Set on by STD.
OF	overflow flag. Indicates signed overflow. True if the high order (sign) bit was changed by overflow.

Address Expressions

Address expressions follow normal 8086/88 rules. For example:

```
[234]
DS:[0]
[BP+98]
variable
variable+22
variable[22]
variable[BP+22]
ES:variable[BP]+22
```

Address Typing

If an instruction includes a register, the type of the register determines the type of the operation. If no register is present, the type of a variable is used. If neither is present or the type of the variable is incorrect, the key-words BYTE, WORD, DWORD, QWORD or TBYTE must be used. BYTE means the operand has a length of one byte, WORD means two bytes, DWORD means four bytes, QWORD means eight bytes and TBYTE means ten bytes.

Examples:

```
MOV      [44],AX
MOV      FOO,1
INC      WORD ES:[BX]
FMUL     QWORD [BP+22]
```

Comments

A non-quoted semi-colon causes the rest of a line to be ignored.

Assembler Directives

Directives may be in either upper or lower case.

Cseg: `CSEG`
Dseg: `DSEG`
Eseg: `ESEG`

The DSEG directive indicates that Small Case data or Large Case static data follows, the CSEG directive indicates that code follows, and the ESEG directive indicates that Large Case array or structure data follow. The default is DSEG. DSEG and CSEG directives may be placed anywhere but all code must follow a CSEG and all data must follow a DSEG or an ESEG.

End: `END`

The END statement is optional and does nothing.

Equate: `identifier EQU expression`

Equates are not evaluated until used so they may contain any sort of expression or mnemonic.

```
LF equ 0aH
PORT equ 201H
```

Even: `EVEN`

Even forces even alignment by inserting a zero byte if required. Words should be on even boundaries on the 8086 for improved performance. On the 8088 it does not make any difference.

If: `IF expression`
Else: `ELSE`
Endif: `ENDIF`

The control directives `IF`, `ELSE`, and `ENDIF` have been added to support conditional assembly. Any symbolic name — set by an `EQU` directive — can be used. For example:

```
IF LARGE_CASE
     mov ax,[bp+6]
ELSE
     mov ax,[bp+4]
ENDIF
```

Include: `INCLUDE "filename"`

The indicated file is included in the source.

Offset: `OFFSET identifier`

`OFFSET` generates the offset of the variable.

Public: `PUBLIC identifier [:BYTE|WORD etc.] [,...]`

Public declares that the listed variables are public. If an identifier is not defined in the file, it is assumed to be external. This allows the same file containing `PUBLIC` declarations to be included in all of the modules of a system.

An identifier may be followed by a colon and the keyword `BYTE`, `WORD`, `DWORD`, `QWORD`, or `TBYTE`. This allows a type to be associated with an external variable. The placement of `PUBLIC` statements is important. They must be in the same segment (`DSEG` or `CSEG`) as the symbols they name. In addition, the `PUBLIC` for a symbol must not follow its definition.

Seg: `SEG identifier`

`SEG` is a Large Case directive which is similar to `OFFSET` except that it generates the segment of the variable rather than the offset.

Reserving Storage

Bytes, words, double-words and quad-words are declared with the DB, DW, DD and DQ directives.

```
[label[:]] DB | DW | DD | DQ value [,value]...
```

Values are truncated to bytes within DB, words within DW and double-words within DD. The exception is the form

```
DB 'string of any length',0
```

DD values may be binary (without a period or 'E' exponent), single precision floating point, or a Large Case pointer. DQ values are always floating point.

Storage can be reserved with RB and RW.

```
[label[:]] RB or RW expression
```

Reserves the indicated number of bytes or words. They are initialized to zero at run time. Caution: RB's and RW's are moved to high memory so they will not be adjacent to the DB's, DW's, DD's, and DQ's they are declared next to.

The Large Case `@` operator creates a long (4 byte) pointer in DSEG and returns its offset. `@` is normally used with `LES` to load a long (4 byte) pointer to a variable.

Differences Between MASM86 and ASM88.

1. Code Macros, MPL, SEGMENT etc. are missing.
2. The public label MAIN_ must be declared somewhere in a program. It identifies the initial entry point.
3. Jump optimization is performed. This means that the assembler assembles JMP as a two byte jump when possible and that jump relative to an address over 128 bytes away is turned into a jump around a jump.For example, a JZ to a label more than 128 bytes away would become a JNZ around a JMP.
4. DQ's values are always floating point.
5. Eight byte binary is not supported.
6. The word 'POINTER' (or 'PTR') is not used. An anonymous variable is 'WORD [BX]' instead if 'WORD PTR [BX]'. The mnemonics LCALL, LJMP and LRET are used for the long forms of CALL, JMP and RET.

8086 Instructions

Elements of Instructions

The following is a description of the various types of operands:

reg	Any general or byte register can be used.
breg	Any byte register.
wreg	Any general register.
segreg	Any segment register.
rm	A memory reference.
regrm	Any general register or memory reference.
constant	A constant expression.
label	The label of a statement.

Instructions

AAA ASCII Adjust forAddition — changes the contents of AL to valid unpacked decimal number; the high-order nybble is zero. Updates AF and CF. OF, PF, SF, and ZF are undefined after execution.

AAD ASCII Adjust for Division — AH is multiplied by 10 and added to AL. Updates PF, SF, and ZF. AF, CF, and OF are undefined after execution.

AAM ASCII Adjust for Multiply — AL is divided by 10. The result goes in AH and the remainder into AL. Updates PF, SF, and ZF. AF, CF, and OF are undefined after execution.

AAS ASCII Adjust for Subtraction. Repairs AL when AL is the result of ASCII subtraction. Updates AF and CF. OF, PF, SF, and ZF are undefined after execution.

ADC Adds the right operand and the carry bit to the left operand. Updates all the flags.

```
ADC AX | AL,constant
ADC regrm,reg | constant
ADC reg,regrm
```

```
adc  ax,ax
adc  al,harry[bp+55]
adc  word [bp+5],0
adc  ax,ax
```

ADD Adds the right operand to the left operand. Updates AF, CF, OF, PF, SF and ZF.

ADD AX | AL,constant
ADD regrm,reg | constant
ADD reg,regrm

```
add  ax,ax
add  al,harry[bp+55]
add  word [bp+5],0
```

AND Logically "and"s the right operand to the left operand. Updates CF, OF, PF, SF, and ZF. AF is undefined after execution.

AND AX | AL,constant
AND regrm,reg | constant
AND reg,regrm

```
and  ax,dx
and  al,harry[bp+55]
and  word [bp+5],0FH
```

CALL Pushes the address of the next instruction and jmps to the indicated address. Call's can be direct to a label or indirect through a word register or a word in memory. No flags are affected.

CALL label | regrm

```
call laba
call bx
call word es:[bx]
```

CBW sign extend AL into AX. No flags are affected.

CLC clear carry flag.

CLD clear direction flag.

CLI clear interrupt enable flag. Disables interrupts.

CMC complement carry flag.

CMP Compares operands. All flags are affected.

CMP AX | AL,constant
CMP reg,regrm

```
cmp  ax,ax
cmp  al,harry[bp+55]
cmp  word [bp+5],0
```

CMPSB compare byte at DS:SI TO ES:DI. Increment SI and DI.
CMPSW compare word at DS:SI to ES:DI. Add 2 to SI and DI.

If the direction flag is on, registers are decremented instead of incremented. These instructions are usually used with a REP, REPZ or REPNZ prefix. All flags are affected.

CWD sign extend AX into DX:AX. No flags are affected.

DAA Decimal Adjust for Addition. Adjusts AL after packed addition. Updates AF, CF, PF, SF, and ZF. OF is undefined after execution.

DAS Decimal Adjust for Subtraction. Adjusts AL after packed subtraction. Updates AF, CF, PF, SF, and ZF. OF is undefined after execution.

DEC Decrements the operand. Updates AF, OF, PF, SF, and ZF.

DEC wreg | regrm

```
dec  di
dec  bl
dec  chr
```

DIV Divide AX by byte operand with result in AL and remainder in AH or divide DX:AX by word operand with result in AX and remainder in DX. AF, CF, OF, PF, SF, and ZF are undefined after execution.

DIV regrm

```
div  cx
```

ESC triggers the 8087. If there is no 8087, this instruction should not be used. The constant/8 is added to the esc instruction. The constant mod 8 is the middle 3 bits of the r/m. No flags are affected.

ESC constant,rm

HLT stops the processor. The processor stops until an external interrupt occurs. No flags are affected.

IDIV Integer divide AX by byte operand with result in AL and remainder in AH or integer divide DX:AX by word operand with result in AX and remainder in DX. AF, CF, OF, PF, SF, and ZF are undefined after execution.

IDIV regrm

IMUL Integer multiply AL by byte operand with result in AX or integer multiply AX by word operand with result in DX:AX. Updates CF and OF. AF, PF, SF, and ZF are undefined after execution.

IMUL regrm

IN input from a port into AL or AX. A constant port must be in the range 0 to 255. The use of DX for port allows addressing all 65535 ports. No flags are affected.

IN AL | AX,constant
IN AL | AX,DX

```
in   al,44
in   ax,dx
```

INC Increment the operand. Updates AF, OF, PF, SF, and ZF.

INC wreg | regrm

```
inc  di
inc  chr
```

INT cause a software interrupt. The int instruction causes the execution of the associated interrupt routine. Interrupts are the usual way to call the operating system from the assembler. An interrupt pushes the flags, pushes CS, pushes IP disables interrupts and LJMP's to the address at 0:interrupt number times 4. The constant must be in the range 0 to 255.

Interrupt 3 generates a one byte instruction. Debuggers use interrupt 3 for breakpoints. A program run under DEBUG can use an 'int 3' to call the debugger. Updates IF and TF.

INT constant

```
int  0C1H
int  3
```

INTO	interrupt on overflow. Cause an interrupt 4 if the overflow bit is set. No flags are affected.
IRET	return from an interrupt. Flags are restored from stack.

JA/NBE	jump on above/not below or equal.
JAE/NB	jump on above or equal/not below.
JB/NAE	jump on below/not above or equal.
JBE/NA	jump on below or equal/not above.
JC	jump on carry.
JCXZ	jump if CX is not equal to zero.
JE/Z	jump on equal/zero.
JG/NLE	jump on greater/not less than or equal.
JGE/NL	jump on greater than or equal/not less.
JL/NGE	jump on less/not greater than or equal.
JLE/NG	jump on less or equal to/not greater.
JMP	jump unconditionally.
JNC	jump on not carry.
JNE/NZ	jump on not equal/not zero,
JNO	jump on not overflow.
JNS	jump on not sign (positive).
JNP/PO	jump on not parity/parity odd.
JO	jump on overflow.
JP/PE	jump on parity/parity equal.
JS	jump on sign (negative).

The words 'above' and 'below' refer to unsigned comparisons. The words 'greater' and 'less' refer to signed comparisons.

ASM88 will turn a jump relative into the five byte equivalent if the target is out of range. No flags are affected.

Jmp's to a label will generate either the two or three byte form depending upon the distance of the label. Jmp's can be direct to a label or indirect through a word register or a word in memory.

JMP label | regrm

```
jmp  laba
jmp  bx
jmp  word es:[bx]
jmp  laba[bx]
```

LAHF load AH from flags. No flags are affected. The format of AH is:

SF	ZF	X	AF	X	PF	X	CF

LCALL long call. LCALL pushes the CS, pushes the instruction pointer, and does a long jump indirect through memory. The memory must contain two words: the new instruction pointer and the new CS. No flags are affected.

LCALL rm

```
lcall laba[bx]
```

LDS loads a register (usually an index register - BX,SI or DI) and DS. It is used to form a long pointer so that data outside of DSEG and CSEG can be addressed. No flags are affected.

LDS wreg,regrm

```
lds  bx,vara
```

LEA loads the offset of the referenced memory location into a register. No flags are affected.

LEA wreg,rm

```
lea  ax,[si+di+44]
lea  ax,vara
mov  ax,offset vara   ; same effect as above
```

LES

loads a register (usually an index register - BX,SI or DI) and ES. It is used to form a long pointer so that data outside of DSEG and CSEG can be addressed. No flags are affected.

LES wreg,regrm

```
les  di,vara
```

LJMP

long jump. Ljmp's can only be indirect through memory. The memory must contain two words: the new instruction pointer and the new CS. No flags are affected.

LJMP label

LOCK

Lock the bus. LOCK demands a bus lock for the following instruction. Usually used with XCHG to implement semaphores. No flags are affected.

LOCK instruction

```
     mov  al,1
lock xchg laba,al
```

LODSB load byte at DS:SI into AL. Increment SI.
LODSW load word at DS:SI into AX. Add 2 to SI.

If the direction flag is on, registers are decremented instead of incremented. These instructions are usually used with a REP, REPZ or REPNZ prefix. No flags are affected.

```
lodsb
rep lodsw
```

LOOP	decrement CX and jump if CX not equal to zero.
LOOPE/ZZ	decrement CX and jump if CX not zero and the zero flag is set.
LOOPNE/Z	decrement CX and jump if CX not zero and the zero flag is cleared.

LOOP, LOOPZ all decrement CX, check it for zero and if not zero, do the jump. LOOPZ and LOOPNZ also check the zero flag. No flags are affected.

LOOP label
LOOPE/Z label
LOOPNE/Z label

LRET perform a long return. Assumes the procedure was called with an LCALL. Both the instruction pointer and the new CS must be on the stack. The optional constant is added to SP after the return address is removed. Languages other than C use this to remove parameters from the stack. C has the caller remove parameters so that a variable number of parameters can be supported. No flags are affected.

LRET | constant

MOV The contents of the right operand are moved to the left operand. No flags are affected.

MOV segreg,regrm
MOV regrm,segreg | reg
MOV reg,constant | regrm
MOV rm,constant

```
mov  ax,bx
mov  cx,ds
mov  es,cx
mov  vara,ax
mov  si,vara
mov  bl,varb[si+di]
```

MOVSB move byte from DS:SI to ES:DI. Increment/decrement SI and DI.
MOVSW move word from DS:SI to ES:DI. Add/subtract 2 to/from SI and DI.

If the direction flag is on, registers are decremented instead of incremented. These instructions are usually used with a REP, REPZ or REPNZ prefix. No flags are affected.

```
     movsb
rep  movsw
```

MUL Multiply AL by byte operand with result in AX or multiply AX by word operand with result in DX:AX. Updates CF and OF. AF, PF, SF, and ZF are undefined after execution.

MUL regrm

```
mul  CX
mul  vara
```

NEG Negate the operand. Updates AF,CF, OF, PF, SF, and ZF.

NEG regrm

```
neg  ax
neg  vara
```

NOP do nothing in three cycles. No flags are affected.

NOT Invert the bits of the operand. No flags are affected.

NOT regrm

```
not  ax
```

OR logical or of the operands. Updates CF, OF, PF, SF, and ZF. AF is undefined after execution.

OR AX | AL,constant
OR regrm,constant | reg
OR reg,regrm

```
or   ax,ax
or   al,harry[bp+55]
```

OUT Output a byte or word to a port. A constant port must be in the range 0 to 255. The use of DX for port allows addressing all 65535 ports. No flags are affected.

OUT constant,AL | AX
OUT DX,AL | AX

```
out  dx,ax
out  33,al
```

POP The word contents of SS:SP are moved to the operand and the stack pointer is incremented by 2. CS cannot be popped as this would kill the system. No flags are affected.

POP wreg | regrm | segreg

```
pop  ax
pop  total
pop  word es:[bx]
```

POPF The flags are popped off of the stack.

PUSH Two is subtracted from SP and the word operand is moved to SS:SP. No flags are affected.

PUSH wreg | regrm | segreg

```
push ax
push total
```

PUSHF The flags are pushed onto the stack. No flags are affected.

RCL rotate left through carry. The carry bit ends up as the new low bit and the high bit becomes the carry bit. Updates CF and OF.

RCL regrm,1 | CL

RCR rotate right through carry. The carry bit ends up as the new high bit and the low bit becomes the carry bit. Updates CF and OF.

RCR regrm,1 | CL

REP decrement CX on each iteration and continue while not zero.

REPZ decrement CX on each iteration and continue while CX is not zero and the zero flag is on.

REPNZ decrement CX on each iteration and continue while CX is not zero and the zero flag is off.

These prefixes can only be used with the string instructions; they cause the string instruction to be repeated. No flags are affected.

REP instruction
REPZ instruction
REPNZ instruction

```
rep  movsb
repz stosw
```

RET Return from a call. Only the instruction pointer is on the stack. The optional constant is added to SP after the return address is removed. Languages other than C use this to remove parameters from the stack. C has the caller remove parameters so that a variable number of parameters can be supported. No flags are affected.

RET | constant

```
ret  5
ret
```

ROL rotate left. The high bit ends up in carry and as the new low bit. Updates CF and OF.

ROL regrm,1 | CL

ROR rotate right. The low bit ends up in carry and as the new high bit. Updates CF and OF.

ROR regrm,1 | CL

SAHF

New flags are loaded from AH. Updates AF, CF, PF, SF, and ZF. The format of AH is:

SF	ZF	X	AF	X	PF	X	CF

SAL	shift arithmetic left. The high bit goes to carry and the new low bit becomes zero.
SHL	shift left. The high bit goes to carry and the new low bit becomes zero.

Updates CF, OF, PF, SF, and ZF. AF is undefined after execution.

SAL regrm,1 | CL
SHL regrm,1 | CL

SAR shift arithmetic right. The low bit becomes the carry bit, the high bit is left alone (i.e. the sign remains the same). Updates CF, OF, PF, SF, and ZF. AF is undefined after execution.

SAR regrm,1 | CL

SBB Subtract the right operand and the carry bit from the left operand. Updates AF, CF, OF, PF, SF, and ZF.

SBB AX | AL,constant
SBB regrm,constant | reg
SBB reg,regrm:>:nl.

```
sbb  ax,ax
sbb  al,harry[bp+55]
sbb  word [bp+5],0
sbb  ax,ax
```

SCASB	compare AL to byte at ES:DI. Increment DI.
SCASW	compare AX to word at ES:DI. Add 2 to DI.

If the direction flag is on, registers are decremented instead of incremented. These instructions are usually used with a REP, REPZ or REPNZ prefix. Updates AF, CF, OF, PF, SF, and ZF.

SHR shift right. The low bit goes to carry, the new high bit is zero. Updates AF, CF, OF, PF, SF, and ZF.

SHR regrm,1 | CL

```
shr  al,1
mov  cl,4
shr  vara,cl
```

STC	set the carry flag.
STD	set the direction flag.
STI	set interrupts enabled.
STOSB	store AL at ES:DI. Increment DI.
STOSW	store AX at ES:DI. Add 2 to DI.

If the direction flag is on, registers are decremented instead of incremented. These instructions are usually used with a REP, REPZ or REPNZ prefix. No flags are affected.

SUB Subtracts the right operand from the left operand. Updates AF, CF, OF, PF, SF and ZF.

SUB AX | AL,constant
SUB regrm,reg | constant
SUB reg,regrm

```
sub  ax,ax
sub  al,harry[bp+55]
sub  word [bp+5],0
```

TEST logically ands the operands. The operands are unchanged. Updates CF, OF, PF, SF, and ZF. AF is undefined after execution.

TEST reg,constant | regrm
TEST ax,constant
TEST regrm,constant | reg

```
test al,1
test ax,80h
test chr,44h
test ax,vara
test vara,ax
```

WAIT halts the processor until the 8087 is ready for an instruction. No flags are affected.

XCHG The contents of the two operands are exchanged. XCHG is often used to implement semaphores. No flags are affected.

XCHG AX,reg
XCHG reg,regrm
XCHG regrm,reg

```
xchg ax,bx
xchg al,ah
xchg vara,si
```

XLAT Move the contents of the byte at BX+AL into AL. No flags are affected.

XLAT

XOR Logically "exclusive or"s the right operand to the left operand. Updates CF, OF, PF, SF, XOR ZF. AF is undefined after execution.

XOR AX | AL,constant
XOR regrm,reg | constant
XOR reg,regrm

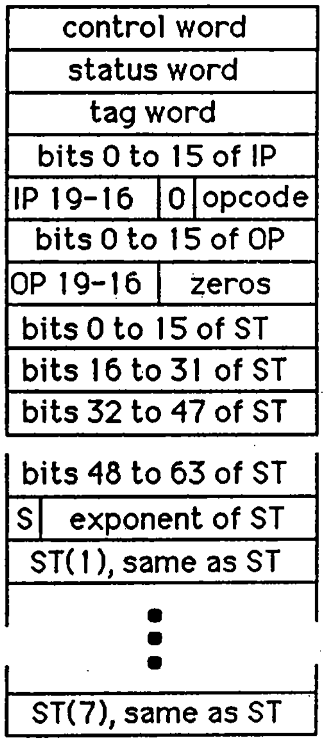
```
xor  ax,dx
xor  al,harry[bp+55]
xor  word [bp+5],0FH
```

8087

The 8087 is the numerics co-processor for the 8086 and 8088. It extends the 8086 architecture by adding instructions for fast and accurate floating point operations. Adding an 8087 to an IBM PC or other 8088 or 8086 based computer that has provision for an 8087 is usually as simple as purchasing the chip and plugging it in.

The 8087 contains an eight element stack. The stack top is referred to as 'ST'. Other elements are referred to as 'ST(i)' where i is between 0 and 7 and is the index of the element. ST(0) is the same as ST. The usual use of the floating point stack is to push two elements and then do a binary operation on them but there are several variations on instruction types. Each element of the stack is maintained as an 80 bit extended precision value. The extra precision minimizes round off errors.

The 8087 context includes both the floating point stack and three status registers. The entire context, as saved by FSAVE and restored by FRSTOR is:



IP stands for instruction pointer and is the 20 bit address of the last instruction. OP is the 20 bit address of the last operand referenced. S is the sign bit.

The portion of the state other than the eight stack elements is called the environment and can be loaded with FLDENV and stored with FSTENV.

Control Word

The control word can be loaded with FLDCW and stored with FSTCW and has the following format:

X	X	X	IC	RC	PC	IEM	X	PM	UM	OM	ZM	DM	IM

X	reserved.
IC	infinity control. 0 is projective which is default. 1 is affine.
RC	rounding control. 0 is round to even (default). 1 is round down. 2 is round up. 3 is truncate.
PC	precision control. 0 is single precision, 1 is double precision and 2 is full precision which is default.
IEM	interrupts enable mask. 0 means disabled which is default.
PM	precision exception mask. All masks are default 1 which means apply the chip default action. A zero means the exception should trigger a user written exception handler procedure.
UM	underflow exception mask.
OM	overflow exception mask.
ZM	zero exception mask.
DM	denormalized exception mask.
IM	invalid operation exception mask.

Status Word

The status word has the following format:

B	C3	ST	C2	C1	C0	IR	X	PE	UE	OE	ZE	DE	IE

B	busy. One if 8087 is executing an instruction.
C	C3,C2,C1,C0 are the completion codes. These are discussed below.
ST	index of stack top element.
IR	interrupt request. On if an 8087 interrupt is pending.
PE	precision exception.
UE	underflow exception.
OE	overflow exception.
ZE	zero divide exception.
DE	denormalized exception.
IE	invalid operation exception.

Tag Word

The tag word has the following format:

tag(7)	tag(6)	tag(5)	tag(4)	tag(3)	tag(2)	tag(1)	tag(0)

tag = 00 if valid,
01 if zero,
10 if not a number, infinity or unnormal, or
11 if empty.

Condition Codes

Following an FCOM (compare), the condition codes are:

C3	C2	C0	
0	0	0	ST > source.
0	0	1	ST < source.
1	0	0	ST == source.
1	1	1	the relationship is unknown.

The status word is arranged so the following code sequence may be used.

```
FSTSW STAT            ;store the 8087 status word
FWAIT                 ;wait for the store
MOV   AH,BYTE STAT+1  ;load hi byte of status into AH.
SAHF                  ;load flags from AH.

JB...                 ;jump if ST < source
JBE...                ;jump if ST <= source
JA...                 ;jump if ST > source
JAE...                ;jump if ST >= source
JE...                 ;jump if ST == source
JNE...                ;jump if ST != source.
```

The FXAM instruction shows if the stack top is an infinity or unnormal.

C3	C2	C1	C0		
0	0	0	0	+	unnormal.
0	0	0	1	+	not a number.
0	0	1	0	-	unnormal.
0	0	1	1	-	NAN.
0	1	0	0	+	normal.
0	1	0	1	+	infinity.
0	1	1	0	-	normal.
0	1	1	1	-	infinity.
1	0	0	0	+	zero.
1	0	0	1		empty.
1	0	1	0	-	zero.
1	0	1	1		empty.
1	1	0	0	+	denormalized.
1	1	0	1		empty.
1	1	1	0	-	denormalized.
1	1	1	1		empty.

8087 Instructions

w stands for 16 bit word, d stands for 32 bit short, q stands for 64 bit quad word and i stands for an index in the range of 0 to 7.

F2XM1 ST = 2**ST-1.

```
f2xm1
```

FABS ST = absolute value(ST)

```
fabs
```

FADD add real.

```
fadd                    ;ST(1)=ST(1)+ST. pop stack.
fadd ST,ST(i)
fadd ST(i),ST
fadd d
fadd q
```

FADDP add real and pop the stack.

```
faddp ST(i),ST
```

FBLD push a BCD operand onto the stack.

```
fbld q
```

FBSTP store and pop a BCD value.

```
fbstp q
```

FCHS change the sign of the stack top

```
fchs
```

FCLEX
FNCLEX clear 8087 exceptions. The 'N' form has no WAIT.

```
fclex
fnclex
```

FCOM compare reals.

```
fcom                 ;compare ST to ST(1)
fcom ST(i)           ;compare ST to ST(i)
fcom d               ;compare ST to float
fcom q               ;compare ST to double
```

FCOMP compare real and pop stack.

```
fcomp                ;compare ST to ST(1)
fcomp ST(i)          ;compare ST to ST(i)
fcomp d              ;compare ST to float
fcomp q              ;compare ST to double
```

FCOMPP compare real and pop stack twice.

```
fcompp               ;compare ST : ST(1). pop both.
```

FDECSTP increment stack top pointer.

```
fdecstp
```

FDISI
FNDISI disable interrupts. The 'N' form does not WAIT

```
fdisi
fndisi
```

FDIV real divide.

```
fdiv           ;ST(1)=ST(1)/ST. pop stack.
fdiv ST,ST(i)
fdiv ST(i),ST
fdiv d
fdiv q
```

FDIVP real divide and pop the stack.

```
fdivp ST(i),ST
```

FDIVR real reverse divide.

```
fdivr                ;ST(1)=ST/ST(1).
fdivr ST,ST(i)
fdivr ST(i),ST
fdivr d
fdivr q
```

FDIVRP real reverse divide and pop the stack.

```
fdivrp ST(i),ST
```

FENI
FNENI enable 8087 interrupts. The 'N' form does not WAIT.

```
feni
fneni
```

FFREE free an 8087 stack element.

```
ffree ST(i)
```

FIADD add an integer to the top os stack

```
fiadd w                                    ;add an 8086
word
fiadd d                                    ;add a long
```

FICOM compare integer to top of stack.

```
ficom w              ;compare to 8086 word
ficom d              ;compare to a long
```

FICOMP compare integer to top of stack and pop.

```
ficomp w             ;compare to 8086 word
ficomp d             ;compare to a long
```

FIDIV divide top of stack by integer..

```
fidiv w              ;divide by 8086 word
fidiv d              ;divide by a long
```

FIDIVR ST = integer / ST.

```
fidivr w            ;divide 8086 word by ST
fidivr d            ;divide a long by ST
```

FILD push an integer.

```
fild w              ;load an 8086 word
fild d              ;load a long
fild q              ;load an 8 byte integer
```

FIMUL multiply ST by an integer.

```
fimul w             ;multiply by an 8086 word.
fimul d             ;multiply by a long
```

FINCSTP increment the stack pointer.

```
fincstp
```

FINIT
FNINIT initialize the 8087. This instruction should precede any other 8087 instruction in a program. The 'N' form does not WAIT.

```
finit
fninit
```

FIST store an integer.

```
fist w              ;store an 8086 word.
fist d              ;store a long
```

FISTP store an integer and pop the stack.

```
fistp w             ;store an 8086 word.
fistp d             ;store a long
```

FISUB subtract an integer from top of stack.

```
fisub w             ;subtract 8086 word
fisub d             ;subtract long
```

FISUBR ST = integer - ST.

```
fisubr w            ;subtract ST from 8086 word
fisubr d            ;subtract ST from long
```

FLD push a floating point value.

```
fld  ST(i)
fld  d
fld  q
fld  tbyte t
```

FLDCW load processor control word

```
fldcw w
```

FLDENV load 8087 environment from memory.

```
fldenv env
```

FLDLG2 load log base 10 of 2.

```
fldlg2
```

FLDLN2 load log base e of 2.

```
fldln2
```

FLDL2E load log base 2 of e.

```
fldl2e
```

FLDL2T load log base 2 of 10.

```
fldl2t
```

FLDPI load PI.

```
fldpi
```

FLDZ load zero.

```
fldz
```

FLD1 load one.

```
fld1
```

FMUL real multiply.

```
fmul                ;ST(1)=ST(1)*ST. pop stack.
fmul ST,ST(i)
fmul ST(i),ST
fmul d
fmul q
```

FMULP multiply real and pop the stack.

```
fmulp ST(i),ST
```

FNOP no operation..

```
fnop
```

FPATAN partial arctangent.

```
fpatan
```

FPREM partial remainder.

```
fprem
```

FPTAN partial tangent

```
fptan
```

FRNDINT round to integer.

```
frndint
```

FRSTOR restore 8087 state

```
frstor state
```

FSAVE
FNSAVE save entire 8087 state. The 'N' form does not WAIT.

```
fsave state
fnsave state
```

FSCALE binary scale ST by ST(1).

```
fscale
```

FSQRT take square root of ST.

```
fsqrt
```

FST store real.

```
fst  ST(i)
fst  d
fst  q
```

FSTCW
FNSTCW store control word. The 'N' form does not WAIT.

```
fstcw w
fnstcw w
```

FSTENV
FNSTENV store the 8087 environment. The 'N' form does not WAIT.

```
fstenv env
fnstenv env
```

FSTP store real and pop.

```
fstp ST(i)
fstp d
fstp q
fstp tbyte t
```

FSTSW
FNSTSW store status word. The 'N' form does not WAIT.

```
fstsw w
fnstsw w
```

FSUB subtract real.

```
fsub              ;ST(1)=ST(1)-ST. pop stack.
fsub ST,ST(i)
fsub ST(i),ST
fsub d
fsub q
```

FSUBP real subtract and pop the stack.

```
fsubp ST(i),ST
```

FSUBR real reverse subtract.

```
fsubr             ;ST(1)=ST-ST(1).
fsubr ST,ST(i)
fsubr ST(i),ST
fsubr d
fsubr q
```

FSUBRP real reverse subtract and pop the stack.

```
fsubrp ST(i),ST
```

FTST compare ST to zero.

```
ftst
```

FWAIT wait for 8087. Same as WAIT.

```
fwait
```

FXAM set condition codes from top of stack.

```
fxam
```

FXCH exchange stack elements.

```
fxch                ;exchange ST and ST(1)
fxch ST(i)
```

FXTRACT decompose into exponent and significand.

```
fxtract
```

FYL2X ST(1) = ST(1) * log 2 ST.

```
fyl2x
```

FYL2XP1 ST(1) = ST(1) * log 2 (ST+1).

```
fyl2xp1
```